# oneAPI for NVIDIA* GPU 2025.0.0 ガイド

この記事は、Codeplay 社の許可を得て iSUS (IA Software User Society) が作成した 2024 年 11 月 8 日時点の『oneAPI for NVIDIA® GPUs 2025.0.0』の日本語参考訳です。原文は更新される可能性があります。原文と翻訳文の内容が異なる場合は原文を優先してください。

---

以前のバージョンのガイド: **2023.0.0** | **2023.1.0** | **2023.2.1** | **2024.0.0** | **2024.2.0**
**『oneAPI for AMD* GPU ガイド』はこちら**


oneAPI for NVIDIA* GPU は、開発者が DPC++/SYCL* を利用して oneAPI アプリケーションを作成し、それらを NVIDIA* GPU 上で実行できるようにするインテル® oneAPI ツールキット向けのプラグインです。

このプラグインは、CUDA* バックエンドを DPC++ 環境に追加します。このドキュメントでは、「oneAPI for NVIDIA* GPU」と「DPC++ CUDA* プラグイン」は同じ意味で使われています。

oneAPI の詳細については、インテル® oneAPI の概要 (英語) を参照してください。

oneAPI for NVIDIA* GPU の使用を開始するには、「導入ガイド」を参照ください。

**導入ガイド**

- oneAPI for NVIDIA* GPU のインストール
- DPC++ を使用して NVIDIA* GPU をターゲットにする
- DPC++ のリソース
- SYCL* のリソース
- SYCL* アプリケーションのデバッグ
- MPI ガイド

**パフォーマンス・ガイド**

- はじめに
- プログラミング・モデル
- 最適化の目的
- パフォーマンス解析
- NVIDIA* GPU 上のパフォーマンス
- CUDA* バックエンドの共有 (管理) USM
- 一般的な最適化

**サポート**

- 機能
- 更新履歴
- トラブルシューティング
- ライセンス (英語)

# 導入ガイド

## oneAPI for NVIDIA* GPU のインストール

このガイドでは、DPC++ と DPC++ CUDA* プラグインを使用して、NVIDIA* GPU で SYCL* アプリケーションを実行する方法を説明します。

DPC++ に関連する一般的な情報は、「DPC++ のリソース」の節を参照してください。

### サポートされるプラットフォーム

このリリースは各種 NVIDIA* GPU と CUDA* バージョンで動作するはずですが、Codeplay は評価されていないプラットフォームでの正常な動作を保証するものではありません。

このリリースは、次のプラットフォームで動作します。

| オペレーティング・システム | アーキテクチャー | CUDA* バージョン |
|---|---|---|
| Linux*(glibc 2.31+) | sm_60+、sm_5x (非推奨) | 11.7+ |
| Windows* | sm_60+、sm_5x (非推奨) | 12.0+ |

次のプラットフォームで検証されています。

| オペレーティング・システム | テストされた GPU | CUDA* バージョン |
|---|---|---|
| Linux*(glibc 2.31+) | NVIDIA* A100 40GB (sm_80) | 11.7、12.2 |
| Windows* | NVIDIA* GeForce* RTX 4060 Ti (sm_89) | 12.5 |

### 要件

1. インテル® oneAPI DPC++/C++ コンパイラーおよび必要な依存関係を含むインテル® oneAPI ベース・ツールキット 2025.0.0 をインストールします。
    - インテル® oneAPI ベース・ツールキットは、多くの利用環境に適用できます。
    - ツールキットはバージョン 2025.0.0 である必要があります。それ以外の場合、NVIDIA* GPU 用のプラグインをインストールできません。
2. 「Linux* 向けの NVIDIA* CUDA* インストール・ガイド」(英語) の手順に従って、NVIDIA* GPU ドライバーと CUDA* ソフトウェア・スタックをインストールします。

C++ 開発ツールもインストールする必要があることに注意してください。

- **Linux*:** `cmake`、`gcc`、`make`

  oneAPI アプリケーションをビルドして実行するには、C++ 開発ツールの `cmake`、`gcc`、`g++`、`make` および `pkg-config` をインストールする必要があります。

  次のコンソールコマンドは、一般的な Linux* ディストリビューションに上記のツールをインストールします。

  **Ubuntu***

  ```
  $ sudo apt update
  $ sudo apt -y install cmake pkg-config build-essential
  ```

  **Red Hat* と Fedora***

  ```
  $ sudo yum update
  $ sudo yum -y install cmake pkgconfig
  $ sudo yum groupinstall "Development Tools"
  ```

  **SUSE***

  ```
  $ sudo zypper update
  $ sudo zypper --non-interactive install cmake pkg-config
  $ sudo zypper --non-interactive install pattern devel_C_C++
  ```

  次のコマンドで、ツールがインストールされていることを確認します。

  ```
  $ which cmake pkg-config make gcc g++
  ```

  次のような出力が得られるはずです。

  ```
  /usr/bin/cmake
  /usr/bin/pkg-config
  /usr/bin/make
  /usr/bin/gcc
  /usr/bin/g++
  ```

- **Windows*:** MSVC ビルドツール 2022。ビルドツールのみが必要ですが、完全な MSVC インストールでも機能します。

**インストール**

- oneAPI for NVIDIA* GPU のインストーラー (英語) をダウンロードします。
- Linux* では、ダウンロードした自己展開型インストーラーを実行します。

  ```
  $ sh oneapi-for-nvidia-gpus-2025.0.0-cuda-12.0-linux.sh
  ```

  - インストーラーは、デフォルトの場所にあるインテル® oneAPI ツールキット 2025.0.0 のインストールを検索します。インテル® oneAPI ツールキットが独自の場所にインストールされている場合、`--install-dir /path/to/intel/oneapi` でパスを指定します。
  - インテル® oneAPI ツールキットが home ディレクトリー外にある場合、`sudo` を使用してコマンドを実行する必要があります。

- Windows* では、ダウンロードしたインストーラーを起動します。

  - インストーラーはインテル® oneAPI ツールキットのインストールと、インストール・ディレクトリーを検出します。検出できない場合は、oneAPI DPC++ コンパイラーが配置されているディレクトリーを指定する必要があります。

    例: `C:\Program Files (x86)\Intel\oneAPI\compiler\2025.0`

**環境を設定**

1. 実行中のセッションで oneAPI 環境を設定するには、インテルが提供する `setvars.sh` スクリプトを source するか (Linux*)、または `setvars.bat` を実行します (Windows*)。

   例:

   ```
   # Linux:
   . /opt/intel/oneapi/setvars.sh

   # Windows:
   "C:\Program Files (x86)\Intel\oneAPI\setvars.bat"
   ```

   - Windows* では、使用するコマンドプロンプトは (例えば “x64 Native Tools Command Prompt” を使用して) 、MSVC ビルドツールにアクセスできる必要があります。
   - `clang++` などの LLVM ツールにパスを追加するには、`--include-intel-llvm` オプションを使用します。
   - ターミナルを開くたびにこのスクリプトを実行する必要があります。セッションごとに設定を自動化する方法については、「CLI 開発向けの環境変数を設定する」 (英語) など、関連するインテル® oneAPI ツールキットのドキュメントを参照してください。

2. CUDA* ライブラリーとツールが環境内にあることを確認します。

   - `nvidia-smi` を実行します。実行時の表示に明らかなエラーが認められなければ、環境は正しく設定されています。
   - 問題があれば、環境変数を手動で設定します。

     ```
     # Linux
     $ export PATH=/PATH_TO_CUDA_ROOT/bin:$PATH
     $ export LD_LIBRARY_PATH=/PATH_TO_CUDA_ROOT/lib:$LD_LIBRARY_PATH

     # Windows
     $ set PATH="C:\Program Files\NVIDIA GPU Computing
        Toolkit\CUDA\CUDA_VERSION\bin";%PATH%
     ```

### インストールの確認

DPC++ CUDA* プラグインのインストールを確認するには、DPC++ の `sycl-ls` ツールを使用して、SYCL* で利用可能な NVIDIA* GPU があることを確認します。NVIDIA* GPU が利用できる場合、`sycl-ls` の出力に次のような情報が表示されます。

```
[cuda:gpu][cuda:0] NVIDIA CUDA BACKEND, NVIDIA A100-PCIE-40GB 8.0 [CUDA 12.5]
```

- 上記のように利用可能な NVIDIA* GPU が表示されていれば、DPC++ CUDA* プラグインが適切にインストールされ、設定されていることが確認できます。
- インストールや設定に問題がある場合、「トラブルシューティング」の「sycl-ls の出力でデバイスが見つからない場合」を確認してください。
- 利用可能なハードウェアとインストールされている DPC++ プラグインに応じて、OpenCL* デバイス、インテル® GPU、または AMD* GPU など、ほかのデバイスもリストされることがあります。

### サンプル・アプリケーションを実行

1. 次の C++/SYCL* コードで構成される `simple-sycl-app.cpp` ファイルを作成します。

```
#include <sycl/sycl.hpp>

int main() {
  // カーネルコード内で使用する 4 つの int バッファーを作成
  sycl::buffer<int, 1> Buffer{4};

  // SYCL* キューを作成
  sycl::queue Queue{};

  // カーネルのインデックス空間サイズ
  sycl::range<1> NumOfWorkItems{Buffer.size()};

  // キューへコマンドグループ (ワーク) を送信
  Queue.submit([&](sycl::handler &cgh) {

    // デバイス上のバッファーへの書き込み専用アクセサーを作成
    auto Accessor = Buffer.get_access<sycl::access::mode::write>(cgh);

    // カーネルを実行
    cgh.parallel_for<class FillBuffer>(
        NumOfWorkItems, [=](sycl::id<1> WIid) {
          // インデックスでバッファーを埋めます
          Accessor[WIid] = static_cast<int>(WIid.get(0));
        });
  });

  // ホスト上のバッファーへの読み取り専用アクセサーを作成。
  // キューのワークが完了するのを待機する暗黙のバリア
  auto HostAccessor = Buffer.get_host_access();

  // 結果をチェック
  bool MismatchFound{false};
```

```
  for (size_t I{0}; I < Buffer.size(); ++I) {
    if (HostAccessor[I] != I) {
      std::cout << "The result is incorrect for element: " << I
                << " , expected: " << I << " , got: " << HostAccessor[I]
                << std::endl;
      MismatchFound = true;
    }
  }

  if (!MismatchFound) {
    std::cout << "The results are correct!" << std::endl;
  }

  return MismatchFound;
}
```

2. Linux* では `icpx`、Windows* では `icx-cl` を使用してアプリケーションをコンパイルします。

```
# Linux
$ icpx -fsycl -fsycl-targets=nvptx64-nvidia-cuda simple-sycl-app.cpp -o
simple-sycl-app

# Windows
$ icx-cl -fsycl -fsycl-targets=nvptx64-nvidia-cuda simple-sycl-app.cpp -o
simple-sycl-app
```

インストールされている CUDA* のバージョンによっては、次のような警告が表示されることがありますが、これは無視してもかまいません。

```
icpx: warning: CUDA version is newer than the latest supported version 12.1
[-Wunknown-cuda-version]
```

3. アプリケーションを実行します。

```
# Linux
$ ONEAPI_DEVICE_SELECTOR="cuda:*" SYCL_UR_TRACE=1 ./simple-sycl-app

# Windows
$ set ONEAPI_DEVICE_SELECTOR="cuda:*"
$ set SYCL_UR_TRACE=1
$ simple-sycl-app.exe
```

次のような出力が得られます。

```
<LOADER>[INFO]: loaded adapter 0x0xc7a670 (libur_adapter_cuda.so.0)
SYCL_UR_TRACE: Requested device_type: info::device_type::automatic
SYCL_UR_TRACE: Selected device: -> final score = 1500
SYCL_UR_TRACE:   platform: NVIDIA CUDA BACKEND
SYCL_UR_TRACE:   device: NVIDIA GeForce RTX 4060 Ti
The results are correct!
```

これで、oneAPI for NVIDIA* GPU の環境設定が確認でき、oneAPI アプリケーションの開発を開始できます。

以降では、NVIDIA* GPU で oneAPI アプリケーションをコンパイルして実行するための一般的な情報を説明します。

## DPC++ を使用して NVIDIA* GPU をターゲットにする

### NVIDIA* GPU 向けのコンパイル

**注意:** このガイドの例の `icpx` コンパイラーは、Windows* では `icx-cl` に置き換える必要があります。

NVIDIA* GPU 対応の SYCL* アプリケーションをコンパイルするには、DPC++ に含まれる `icpx` (Linux*) または `icx-cl` (Windows*) コンパイラーを使用します。

例:

```
# Linux
$ icpx -fsycl -fsycl-targets=nvptx64-nvidia-cuda sycl-app.cpp -o sycl-app

# Windows
$ icx-cl -fsycl -fsycl-targets=nvptx64-nvidia-cuda sycl-app.cpp -o sycl-app
```

次のフラグが必要です。

- `-fsycl`: C++ ソースファイルを SYCL* モードでコンパイルするようにコンパイラーに指示します。このフラグは暗黙的に C++ 17 を有効にし、SYCL* ランタイム・ライブラリーを自動でリンクします。
- `-fsycl-targets=nvptx64-nvidia-cuda`: NVIDIA* GPU をターゲットとして、SYCL* カーネルをビルドすることをコンパイラーに指示します。

また、次のフラグを使用して、特定の NVIDIA* アーキテクチャー向けの SYCL* カーネルをビルドすることができます。

```
-Xsycl-target-backend=nvptx64-nvidia-cuda --cuda-gpu-arch=sm_80
```

デフォルトではカーネルは sm_50 用にビルドされ、多様なアーキテクチャーで動作しますが、新しい CUDA* 機能の利用は制限されることに注意してください。

利用できる SYCL* コンパイルフラグの詳細は、『DPC++ コンパイラー・ユーザーズ・マニュアル』(英語) を参照してください。すべての DPC++ コンパイラー・オプションの詳細は、『インテル® oneAPI DPC++/C++ コンパイラー・デベロッパー・ガイドおよびリファレンス』の「コンパイラー・オプション」(英語) を参照してください。

### icpx または icx-cl コンパイラーを使用する

`icpx` コンパイラーは、デフォルトで `-O2` と `-ffast-math` オプションを有効にするため、通常の `clang++` ドライバーよりも積極的な最適化を行います。多くの場合、これによりパフォーマンスは向上しますが、一部のアプリケーションでは問題が生じる可能性があります。その場合、`-fno-fast-math` を使用して `-ffast-math` を無効にして、`-O2` 以外の `-O` オプションを指定することで最適化レベルを変更できます。`$releasedir/compiler/latest/linux/bin-llvm/clang++` にある `clang++` ドライバーを直接起動することで、通常の `clang++` の最適化レベルを適用できます。

**複数ターゲット向けのコンパイル**

NVIDIA* GPU をターゲットにするだけではなく、一度のコンパイルで複数のハードウェア・ターゲットで実行できる SYCL* アプリケーションを生成できます。次の例は、NVIDIA* GPU、AMD* GPU、および SPIR* をサポートする任意のデバイス (インテル® GPU など) で実行できるコードを含む単一のバイナリーを生成する方法を示しています。

```
$ icpx -fsycl -fsycl-targets=spir64,amdgcn-amd-amdhsa,nvptx64-nvidia-cuda \
       -Xsycl-target-backend=amdgcn-amd-amdhsa --offload-arch=gfx1030 \
       -Xsycl-target-backend=nvptx64-nvidia-cuda --offload-arch=sm_80 \
       -o sycl-app sycl-app.cpp
```

上記のコマンドは、次のように分解できます。

- `-fsycl` は、C++ ソースファイルを SYCL* モードでコンパイルするようにコンパイラーに指示します。
- `-fsycl-targets=spir64,amdgcn-amd-amdhsa,nvptx64-nvidia-cuda` は、SYCL* デバイスコードの 3 つのターゲットを指定します。ターゲットは汎用で、特定の GPU アーキテクチャー (インテルおよび NVIDIA* ターゲットのみ) の実行中にジャストインタイム (JIT) でコンパイルされるコードを生成します。次のように、汎用コードに加えて、アーキテクチャー固有のデバイスコードを事前 (AOT) に生成するためのフラグを追加することもできます。
- `-Xsycl-target-backend=amdgcn-amd-amdhsa` は、フラグパーサーに次のフラグが `amdgcn-amd-amdhsa` ターゲットのコンパイラー・バックエンドにのみ渡され、他のターゲットには渡されないことを指示します。値なしで `-Xsycl-target-backend` を指定すると、次のフラグがすべての SYCL* デバイスターゲットのコンパイラー・バックエンドに渡されます。
- `--offload-arch=gfx1030` は、AOT コンパイル用の AMD* GPU アーキテクチャー gfx1030 を指定します。
- `-Xsycl-target-backend=nvptx64-nvidia-cuda` は、フラグパーサーに、次のフラグが `nvptx64-nvidia-cuda` ターゲットのコンパイラー・バックエンドにのみ渡されるように指示します。
- `--offload-arch=sm_80` は、AOT コンパイル用の NVIDIA* GPU アーキテクチャー (コンピューティング機能) sm_80 を指定します。

上記の方法でコンパイルされたバイナリーは、次の環境で SYCL* カーネルを正常に実行できます。

- SPIR* コードの JIT コンパイルを備えたインテルの CPU および GPU
- gfx1030 アーキテクチャーの AMD* GPU は、JIT コンパイルなしでバイナリーから直接実行できます
- コンピューティング能力 8.0 の NVIDIA* GPU は、JIT コンパイルなしでバイナリーから直接実行できます
- JIT コンパイルを備えたその他のサポートされている NVIDIA* GPU

インテル・ハードウェアの AOT コンパイルは、AMD および NVIDIA* ターゲットと組み合わせることも可能であり、`spir64_gen` ターゲットと対応するアーキテクチャー・フラグを使用することで実現できます。例えば、上記のアプリケーションをインテル® グラフィックス・アーキテクチャー開発コード名 Ponte Vecchio 用に AOT でコンパイルするには、次のコマンドを使用できます。

```
$ icpx -fsycl -fsycl-targets=spir64,amdgcn-amd-amdhsa,nvptx64-nvidia-cuda \
       -Xsycl-target-backend=spir64_gen '-device pvc' \
       -Xsycl-target-backend=amdgcn-amd-amdhsa --offload-arch=gfx1030 \
       -Xsycl-target-backend=nvptx64-nvidia-cuda --offload-arch=sm_80 \
       -o sycl-app sycl-app.cpp
```

インテルのターゲットには異なるコンパイラー・ツールチェーンが使用されるため、`--offload-arch=<arch>` とは異なる構文 `-device <arch>` が必要であることに注意してください。

コンパイラー・ドライバーは、各ターゲット + アーキテクチャー・ペアのエイリアスターゲットも提供し、コマンドラインを短縮して理解しやすくします。エイリアスにより、`-Xsycl-target-backend` フラグを指定する必要がなくなりました。上記のコマンドは、以下と同等です。

```
$ icpx -fsycl -fsycl-targets=intel_gpu_pvc,amd_gpu_gfx1030,nvidia_gpu_sm_80 \
       -o sycl-app sycl-app.cpp
```

エイリアスの完全なリストは、DPC++ コンパイラー・ユーザーズ・マニュアルに記載されています。

**NVIDIA* GPU で SYCL* アプリケーションを実行**

NVIDIA* ターゲット向けに SYCL* アプリケーションをコンパイルしたら、ランタイムが SYCL* デバイスとして NVIDIA* GPU を選択しているか確認する必要があります。

通常、デフォルトのデバイスセレクターを使用するだけで、利用可能な NVIDIA* GPU の 1 つが選択されます。しかし、場合によっては、SYCL* アプリケーションを変更して、GPU セレクターやカスタムセレクターなど、より正確な SYCL* デバイスセレクターを設定することもあります。

**DPC++ に公開された NVIDIA* デバイスの制御**

DPC++ などの SYCL* プログラミング実装で特定の GPU のみを使用するように強制することが望ましい場合があります。これは、以降で説明するいくつかの環境変数によって可能になります。これらの環境変数により、ユーザーは共有 GPU クラスター内での GPU リソースの共有を制御することもできます。

デバイスセレクター環境変数

環境変数 `ONEAPI_DEVICE_SELECTOR` を設定して、利用可能なデバイスセットを限定することで SYCL* デバイスセレクターを支援できます。例えば、DPC++ CUDA* プラグインでサポートされるデバイスのみを許可するには、次のように設定します。

```
# Linux
$ export ONEAPI_DEVICE_SELECTOR="cuda:*"

# Windows
$ set ONEAPI_DEVICE_SELECTOR="cuda:*"
```

cuda バックエンドからのデバイスのサブセットのみを許可するには、カンマで区切ったリストを使用します。

例:

```
# Linux
$ export ONEAPI_DEVICE_SELECTOR="cuda:1,3"

# Windows
$ set ONEAPI_DEVICE_SELECTOR="cuda:1,3"
```

次の操作により、Devs に 2 つの NVIDIA* デバイスのみが入力されます。

```
std::vector<sycl::device> Devs;
for (const auto &plt : sycl::platform::get_platforms()) {
  if (plt.get_backend() == sycl::backend::ext_oneapi_cuda) {
    Devs=plt.get_devices();
    break;
  }
}
```

ユーザーが nvidia-smi を呼び出すと、Devs[0] と Devs[1] はそれぞれ 1 と 3 でマークされたデバイスに対応します。

ONEAPI_DEVICE_SELECTOR 環境変数の詳細については、インテル® oneAPI DPC++ コンパイラーのドキュメントで「環境変数」(英語) 参照してください。

NVIDIA* デバイスのみが DPC++ に公開されている場合、上記の ONEAPI_DEVICE_SELECTOR の使用方法は、CUDA_VISIBLE_DEVICES 環境変数を設定するのと同じです。

```
# Linux
$ export CUDA_VISIBLE_DEVICES=1,3

# Windows
$ set CUDA_VISIBLE_DEVICES=1,3
```

NVIDIA* GPU のみが使用可能なこの状況では、同じリストをより簡単な方法で作成できます。

```
std::vector<sycl::device> Devs =
    sycl::device::get_devices(sycl::info::device_type::gpu);if
```

ここでは、Devs[0] と Devs[1] は、nvidia-smi によって 1 と 3 とマークされたデバイスに対応します。

## DPC++ のリソース

- インテル® DPC++ の概要 (英語)
- DPC++ 導入ガイド
- DPC++ コンパイラー・ユーザーズ・マニュアル (英語)
- DPC++ コンパイラーとランタイムのアーキテクチャー設計
- DPC++ 環境変数

## SYCL* のリソース

- SYCL* 2020 仕様
- SYCL* アカデミー学習教材 (英語)
- Codingame インタラクティブ SYCL* チュートリアル (英語)
- IWOCL SYCL* トーク (英語)
- 無料の DPC++ 電子書籍 (英語)
- SYCL* の最新ニュース、学習教材、プロジェクトの紹介 (英語)

## SYCL* アプリケーションのデバッグ

この節では、さまざまなデバイスで SYCL* アプリケーションをデバッグするための情報、ヒント、およびポインターについて説明します。

SYCL* アプリケーションのホストコードは、単純に C++ アプリケーションとしてデバッグできますが、カーネルデバッグのサポートやツールは、ターゲットデバイスによって異なる可能性があります。

**注意:** SYCL* アプリケーションに汎用性がある場合、実際のターゲットデバイスではなく、インテルの OpenCL* CPU デバイスなど、豊富なデバッグサポートとツールを備えたデバイスでデバッグしたほうが有用なことがあります。

### インテルの OpenCL* CPU デバイスでのデバッグ

インテルの OpenCL* CPU デバイスを使用した DPC++ アプリケーションのデバッグについては、『インテル® oneAPI プログラミング・ガイド』の「DPC++ と OpenMP* オフロードプロセスのデバッグ」の節を参照してください。

### NVIDIA* GPU 上での DPC++ を使用したコードのデバッグ

DPC++ と NVIDIA* GPU では、`cuda-gdb` を使用して、DPC++ `cuda` バックエンド向けにコンパイルされたカーネルをデバッグできます。インテル® oneAPI ベース・ツールキットに含まれる `oneapi-gdb` ツールは、インテル® プロセッサーのデバッグにのみ対応しているため、NVIDIA* GPU では使用できません。`cuda-gdb` ツールは、NVIDIA からダウンロードされる CUDA* ツールキットの一部であるため、このデバッガーを使用するには、CUDA* がインストールされていることを確認する必要があります。

`cuda-gdb` の詳しい使用法については、`cuda-gdb` のドキュメント (英語) を参照してください。

`cuda-gdb` デバッガーを使用するコードをコンパイルする場合、次のフラグを使用してデバッグシンボルを有効にする必要があります。

```
$ icpx -G -O0 …
```

**注意:** この記事で見られるような予期しないセグメンテーション・エラーや通信エラーが発生した場合、問題を軽減するには、`cuda-gdb` を起動するときに次の環境変数を設定することを推奨します。

```
CUDBG_USE_LEGACY_DEBUGGER=1
```

### コマンドラインからのデバッグ

コマンドラインからデバッガーを起動するには、実行可能ファイルで次のコマンドを使用します。

```
cuda-gdb ./myexecutable
```

デバッガーを使用するためのコマンドとドキュメントは、NVIDIA のウェブサイト (英語) を参照してください。

**注意:** デバッグセッションを開始すると、`cuda-gdb` は oneAPI Python* スクリプトに関するセキュリティーの問題を警告する場合があります。この警告を削除するには、`/home/.config/gdb/cuda-gdb` にファイル

`cuda-gdbinit` を作成し、インテル® oneAPI ベース・ツールキットの gdb Python* スクリプトへのパスを安全なパスリストに追加する行を挿入します。

```
add-auto-load-safe-path /path/to/libsycl.so.6.1.0-gdb.py
```

例えば、`/opt/intel/oneapi/compiler/2023.1.0/linux/lib/libsycl.so.6.1.0-gdb.py` のようになります。

コマンドの例を以下に示します。

```
add-auto-load-safe-path
/opt/intel/oneapi/compiler/2023.1.0/linux/lib/libsycl.so.6.1.0-gdb.py
```

**cuda-gdb を使用した VSCode によるデバッグ**

**注意:** VSCode を SYCL* で動作させる設定方法については、次のガイドを参照してください。

VSCode でデバッガーを使用するには、起動時に `cuda-gdb` コマンドを使用するように VSCode デバッガー構成ファイル `launch.json` を変更する必要があります。また、必要なデバッグシンボルが生成されるように、前述のコンパイルフラグ (`-G` および `-O0`) が VSCode プロジェクトのコンパイル構成ファイル `task.json` で `icpx` コンパイラーへのパラメーターとして含まれていることを確認する必要があります。

VSCode がデバッグセッションを起動するように構成されている場合は、Shift + Ctrl + P キーを押してコマンドを表示し、「`Debug: Select Debug Session`」と入力して、リストから必要なデバッグセッションの種類を選択し、Enter キーを押します。

**VSCode を使用して例にアタッチしてデバッグセッションを開始**

すでに実行中のプログラムにデバッガーをアタッチすることもできます。デバッグビューの VSCode ターミナルパネルを使用して、コマンドライン・プロンプトに次のように入力します。

```
CUDBG_USE_LEGACY_DEBUGGER=1 ./<TheDPC++Executable> &
```

Shift + Ctrl + P キーを押して、[Debug: Start a new session] を選択し、[C/C++: Intel icpx build attached cuda-gdb debug CUDA target] を選択します。VSCode は実行中のプロセスの一覧を表示します。実行可能ファイルの名前で識別されるプロセスを選択します。

VSCode は、デバッグコンソールから `cuda-gdb` コマンドの直接入力をサポートしています。`cuda-gdb` コマンドを入力するには、次に示すように、`-exec` プレフィクスを追加する必要があります。デバッガーコマンドを実行すると、VSCode GUI とコードパネルがそれに応じて更新されます。

```
Thread 1 "DXTC_d" hit Temporary breakpoint 17, main (argc=1, argv=0x7fffffffcc28) at src/dxtc.cpp:694
694       sleep(20);
→ -exec list cuda kernels
list cuda kernels
Function "cuda kernels" not defined.
> -exec list cuda kernels
```

図 1: VSCode デバッグコンソールに `cuda-gdb` コマンドを直接入力

### カーネルコードのデバッグ

カーネル `cgh.parallel_for()` にステップインすることは可能ですが、カーネルコードに到達するには数回ステップアウトする必要があります。

`parallel_for` ステートメントによってカーネル内またはコードにステップインする場合、次の `cuda-gdb` コマンドを使用して、どのカーネルまたはスレッド (カーネルはスレッド) とそのカーネル内のどの行が現在デバッグされているかを判断したり、カーネルがキューに入れられたばかりのプログラム内のポイントを見つけたりできます。

```
info threads
thread
info cuda kernels
```

ホストスレッドがデバッグフォーカスから外れたと判断され、次に示すようにカーネルがインスタンス化されると、デバッグフォーカスをカーネルに移動できます。

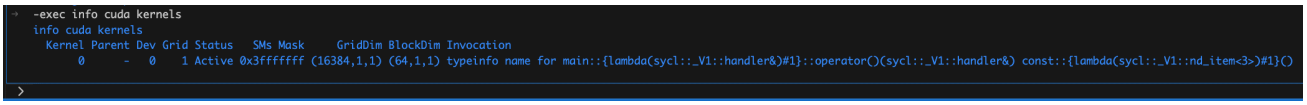


図 2: DPC++ カーネルはキューに投入されているがデバッグフォーカスから外れている

フォーカスを切り替えるには、`cuda-gdb` コマンドを使用します。

```
cuda kernel <id number of the kernel>
```

リストされているカーネルの横に星マークが表示され、カーネルにデバッグフォーカスがあることを示します。

フォーカスを切り替えると、デバッガーは残りの実行中のカーネル (別のスレッド) にあるブレークポイントで停止しますが、デバッガーは SYCL* ライブラリー内の別の場所で停止する可能性があります。この場合、`finish` コマンドを使用してスタックフレームを上に移動し、デバッガーを目的のカーネルコードに戻します。

デバッガーがキューの `wait()` ポイントでホストスレッドに一瞬戻りますが、ステップ実行しているカーネルコードに戻るまで待ちます。

インテル `oneapi-gdb` デバッガーでは、`gdb` の `set scheduler-locking on` または `step` コマンドを使用してデバッガーをロックし、実行中の他のカーネルにランダムにジャンプするのを防ぐことができます。ただし、`cuda-gdb` デバッガーには同等のコマンドがありません。

このような場合、デバッガーが別のカーネルに切り替わったときに、`cuda kernel id` コマンドで目的のカーネルに戻すことができます。デバッガーを使用していて、異なるカーネル (サブグループ) からの複数のバリアに遭遇すると、デバッガーが応答しなくなることがあります。状態を回復するには、デバッガーを強制終了して再起動します。

#### プロセスを強制終了する方法

デバッグセッションがハングアップしたり、何らかの問題が発生した場合、プログラムのプロセスはまだ実行中である可能性があり、新しいデバッグセッションを開始する前に停止する必要があります。プロセスの強制終了は、デバッガー自体から `kill` コマンドを使用して行うこともできます。

Linux* の `ps ux -u <ユーザー名>` コマンドを使用して、実行中のプロセスを一覧表示できます。一覧から、プログラムによって識別されるプロセスを見つけます。コマンドラインで、`kill -9 <プロセス ID 番号>` と入力します。

#### 複数種類のバックエンドを使用

DPC++ が複数のバックエンド (OpenCL* や CUDA* など) をサポートするように設定されているシステムを使用している場合、フィルターを使用してデバッグセッションを適切なバックエンドに関連付ける必要があることがあります。これには、デバッグセッションを開始するときに `SYCL_DEVICE_FILTER` 環境変数を使用します。ターミナル・コマンドラインから、次のコマンドを実行します。

```
SYCL_DEVICE_FILTER=cuda cuda-gdb ./myapp
```

## MPI ガイド

MPI と SYCL* は、選択したバックエンドが両方の実装をサポートしている場合、シームレスに連携できます。プログラマーは、SYCL* と GPU 対応 MPI を組み合わせて、さまざまなバックエンドにわたって 100% 移植可能なコードを作成できます。実際、CPU バックエンドのメモリーを含むあらゆる種類の SYCL* デバイス割り当てメモリーをサポートする MPI を包括するには、より一般的な用語である「デバイス対応」MPI を使用するほうが適切です。

MPI + SYCL* コードを使用する場合、いくつかのバックエンド固有の考慮事項があります。このガイドでは、DPC++ の `cuda` または `hip` バックエンドに関するこのような問題について詳しく説明します。インテル® GPU で GPU 対応 MPI を使用する (`level_zero` バックエンドを使用する) 場合の具体的な問題については、適切なインテルのドキュメントを参照してください。

GPU 対応 MPI を DPC++ の `cuda` または `hip` バックエンドで使用するのは簡単で、ネイティブ `cuda` (`nvcc`) または `hip` (`hipcc`) コンパイラーを使用する方法と似ています。コンパイル方法は、ネイティブ・コンパイラーで使用されるものとほぼ同じです。特定のコンピューティング・クラスターに特化したネイティブ・コンパイラーで GPU 対応 MPI を使用する方法を詳しく説明したドキュメントを参照している場合、命令を `icpx` に移行するのは、ネイティブ・コンパイラー呼び出しを `icpx` に置き換える (および関連する (アーキテクチャー指定子など) コンパイラー・フラグをそれに応じて調整する) のと同じくらい簡単です。次のセクションでは、より完全な詳細を説明します。

### 必要要件

ここでは、cuda バックエンドをサポートするインテル® oneAPI DPC++ コンパイラーが正常にインストール済みであることを前提としています。このバックエンドをサポートする Codeplay oneAPI プラグインのインストール方法については、「oneAPI for NVIDIA* GPU のインストール」を参照してください。また、CUDA* 対応の MPI 実装のビルドも必要です。インテル® oneAPI ツールキットには、CUDA*/ ROCm* 非対応のインテル® MPI 実装が付属していることがあります。その場合は、CUDA* 対応を有効にしてソースからビルドされた OpenMPI または MPICH、または適切なデバイス・アクセラレーションを備えた CRAY* MPICH モジュールなど、別の実装を使用する必要があります。

このガイドで使用されているコード例は、SYCL-samples リポジトリー (英語) リポジトリーで入手できます。

### Codeplay oneAPI プラグインで MPI を使用する

send_recv_buff.cpp (英語) および send_recv_usm.cpp (英語) サンプルコードは、バッファーまたは USM を使用してデバイス対応 MPI を DPC++ で使用する例を示す入門サンプルです。デバイス対応 MPI で sycl::buffer バッファーを使用するには、host_task 内で MPI 呼び出しを行う必要があります。詳細については、send_recv_buff.cpp (英語) サンプルを参照してください。SYCL* USM を MPI で使用するには、常にメインスレッドから MPI 関数を直接呼び出す必要があります。host_tack 内から SYCL* USM を取得する MPI 関数の呼び出しの動作は、現在未定義です。さらに、scatter_reduce_gather.cpp (英語) サンプルでは、MPI を SYCL* 2020 のリダクションおよび parallel_for と併用して、最適化された簡略な複数ランクでのリダクションを実現する方法を示しています。

### MPI ラッパーを使用してコンパイルする

サンプルをコンパイルするには、コンパイラー・ラッパー (mpicxx など) が DPC++ コンパイラー (icpx) を起動できるようにする必要があります。この方法でビルドする方法については、MPI 実装のドキュメントを参照してください。実装によっては、コマンドライン引数または環境変数 (例: MPICH_CXX、OMPI_CXX) を使用して、再構築せずに既定のコンパイラーを変更することもできます。

最初に、パスにラッパーが設定されていることを確認します。

```
$ export PATH=/path/to/your-mpi-install/bin:$PATH
```

次に、サンプルをコンパイルします。

```
$ mpicxx -fsycl -fsycl-targets=TARGET send_recv_usm.cpp -o ./res
```

cuda バックエンドで TARGET を正しく指定する方法の詳細については、「導入ガイド」を参照してください。

サンプルを実行するには 2 つのランクが必要です。次のコマンドで MPI を使用してアプリケーションを実行します。

```
$ mpirun -n 2 ./res
```

-n 2 は、2 つのランクを使用してアプリケーションを実行することを指定します。サンプルに変更を加えるか、各ランクで特別な環境変数を設定しない限り、これは 1 つの GPU を使用して 2 つのランクが実行されることに注意してください。

### Cray* モジュールを使用してコンパイル

一部のクラスターでは、DPC++ と直接リンクできる Cray* MPICH モジュールが利用可能です。これを行うには、通常、ハードウェア固有の Cray* モジュールもロードされていることを確認する必要があります。次に例を示します。

```
$ module load craype-accel-nvidia80
```

その後、`icpx` で直接コンパイルできます。適切な Cray* MPICH ライブラリーをインクルード/リンクする必要があります。リンクするライブラリーは使用するクラスターによって異なるため、適切なドキュメントを参照する必要がありますが、次のようなものになる場合があります。

```
$ icpx -fsycl -fsycl-targets=TARGET send_recv_usm.cpp -o res \
       -I/opt/cray/pe/mpich/8.1.25/ofi/cray/10.0/include/ \
       -L/opt/cray/pe/mpich/8.1.25/ofi/cray/10.0/lib -lmpi -o res
```

次の環境変数の設定が必要な場合もあります。

```
MPICH_GPU_SUPPORT_ENABLED=1
```

その後、特定のクラスターに応じて異なる標準のジョブ送信手順によりプログラムを実行できるようになります。

### 特定のデバイスに MPI ランクを割り当て

単一ノード内の GPU 対応 MPI では、ユーザーは各ランクが特定の GPU を使用するかどうかを制御する機能が必要になります。*`MPI_Send`* などの MPI インターフェイスは、データを受信する対象となる GPU デバイスを指定しません。MPI ランクのみを指定します。HIP および CUDA バックエンドでは、メモリーリークを防ぐため環境変数を使用する必要があります。例えば、各 MPI プロセスと各 GPU の間で 1:1 マッピングを使用する場合は、適切な環境変数を使用して、プロセスごとに 1 つの一意の GPU のみを公開することを推奨します。

各プロセスがローカル MPI ランク ID と一致するデバイス ID にのみ公開するため、環境変数 `ONEAPI_DEVICE_SELECTOR` (英語) または `CUDA_VISIBLE_DEVICES` (英語) を使用する方法の詳細については、導入ガイドの「DPC++ に公開された NVIDIA* デバイスの制御」の節を参照してください。

ローカルランク ID を取得する方法については、MPI 実装のドキュメントを参照してください。

SLURM* システムを利用する場合、GPU アフィニティー・オプション `--gpu-bind` を使用して、`CUDA_VISIBLE_DEVICES` と同様の効果を実現できます。ただし、`--gpu-bind` オプションは計算クラスター固有であるため、クラスターのドキュメントを確認してください。詳細については、SLURM* のドキュメント (英語) 参照してください。

### 制限事項

- DPC++ を使用する CUDA* 対応の MPI では、現在ノード間の MPI の SYCL* 共有 USM をサポートしていません。
- DPC++ を使用した CUDA* 対応 MPI は現在、単一の MPI ランクを単一の GPU にマッピングするか、複数の MPI ランクを単一の GPU にマッピングする場合にのみテストされています。複数の GPU を単一の MPI ランクにマッピングすることはテストされていません。

# パフォーマンス・ガイド

## はじめに

このガイドは、SYCL* プログラミング・モデルと一般的な GPU におけるパフォーマンスの紹介から始まります。次に、GPU でのパフォーマンス解析の基本と、そこで使用される一般的なツールを紹介します。最後に、ベンダー固有の GPU と利用可能なツールについて紹介します。また、無料の書籍『Data Parallel C++』(英語) を一読されることを推奨します。第 15 章では、SYCL* と DPC++ に関連した GPU 上でのパフォーマンスについて説明されています。

NVIDIA* GPU と AMD* GPU の両方に適用される一般的な SYCL* 最適化については、「一般的な最適化」を参照してください。

NVIDIA* GPU をターゲットにする固有の最適化については、「NVIDIA* GPU 上のパフォーマンス」を参照してください。

インテル® GPU 固有のパフォーマンス最適化については、対応するインテル® GPU 固有のパフォーマンス・ガイドを参照してください。

## プログラミング・モデル

グラフィックス処理ユニットは、超並列アーキテクチャーにより、CPU よりも 1 秒あたり多くの浮動小数点演算を実行でき、メモリー帯域幅も高くなっています。これらの機能は、コードの開発時点で GPU アーキテクチャーを使用することを選択した場合にのみ活用できます。

ここでは、GPU における大規模並列処理を表現するプログラミング・モデルが基本となります。SYCL* は OpenCL* や CUDA* と同様のプログラミング・モデルを採用しており、カーネル (GPU によって実行される関数) は work-item によって実行される操作で表現されます。

SYCL* 仕様 (Rev 9) の 3.7.2 節 (英語) では次のように定義されています。

> カーネルが実行のため送信されると、インデックス空間が定義されます。カーネルボディーのインスタンスは、インデックス空間の各ポイントで実行されます。カーネル・インスタンスは work-item (ワーク項目) と呼ばれ、グローバル id を提供するインデックス空間内のポイントで識別されます。それぞれの work-item は同じコードを実行しますが、コードと操作されるデータの実行パスは、work-item のグローバル id を使用して計算を特殊化することで異なります。

SYCL* では、2 つの異なるカーネル実行モデルを利用できます。

SYCL* 仕様 (Rev 9) の 3.7.2.1 節 (英語) では次のように記述されています。

> `range<N>` (`N` は 1、2 または 3) で定義される N 次元のインデックス空間でカーネルを呼び出す単純な実行モデルをサポートします。この場合、カーネルの work-item は独立して実行されます。各 work-item は、タイプ `item<N>` の値によって識別されます。タイプ `item<N>` は、タイプ `id<N>` の work-item 識別子と、カーネルを実行する work-item の数を示す `range<N>` をカプセル化します。

SYCL* 仕様 (Rev 9) の 3.7.2.2 節 (英語) では次のように記述されています。

> work-item を wark-group に編成できる ND-range の実行モデルは インデックス空間よりも粗い粒度の分解を提供します。それぞれの work-group には、work-item で使用できるインデックス空間と同じ次元の work-group id が割り当てられます。work-item には、それぞれ work-group 内で一意のローカル id が割り当てられるため、単一 work-item は、グローバル id、またはローカル id と work-group id の組み合わせで識別できます。特定の work-group 内の work-item は、単一の計算ユニットの処理ユニットで同時に実行されます。SYCL* で使用される work-group は、ND-range と呼ばれます。ND-range は、N 次元のインデックス空間であり、N は 1、2 または 3 です。SYCL* では、ND-range は `nd_range<N>` クラスを介して表現されます。`nd_range<N>` は、グローバルレンジとローカルレンジで構成され、それぞれ `range<N>` タイプの値で表現されます。さらに、タイプ `id<N>` 値で表現されるグローバルオフセットが存在することもありますが、これは SYCL* 2020 では非推奨です。タイプ `nd_range<N>` と `id<N>` は、それぞれ N 要素の整数配列です。`nd_range<N>` で定義される反復回数は、ND-range のグローバルオフセットで開始される N 次元のインデックス空間であり、サイズはグローバルレンジで、ローカル・レンジ・サイズの work-group に分割されます。ND-range の各 work-item は、タイプ `nd_range<N>` の値によって識別されます。タイプ `nd_range<N>` は、グローバル id、ローカル id、および work-group id をすべて `id<N>` (`id<N>` タイプの反復空間オフセットですが、SYCL* 2020 では非推奨) としてカプセル化し、グローバルとローカルレンジを同期して work-group を有効にします。work-group には、work-item のグローバル id と同様の方法で id が割り当てられます。work-item には work-group とゼロからその次元の work-group サイズから 1 を引いた範囲のコンポーネントを保持するローカル id が割り当てられます。つまり、work-group id と work-group 内のローカル id の組み合わせで work-item が一意に定義されます。

work-item は、次の OpenCL* メモリーモデルに従って 3 つの異なるメモリー領域にアクセスできます。

- **グローバルメモリー:** すべての work-group のすべての work-item 間で共有されます。
- **ローカルメモリー:** 同一 work-group のすべての work-item 間で共有されます。
- **プライベート・メモリー:** 各 work-item でプライベートです。

**アーキテクチャー**

SYCL* 仕様では、独立して動作する 1 つ以上の計算ユニット (CU) で構成されるデバイスを考慮することで、OpenCL* 1.2 の仕様に従います。NVIDIA では CU を ストリーミング・マルチプロセッサー (*streaming multiprocessor*) と呼び、AMD では単純に計算ユニット (*compute unit*) と呼んでいます。それぞれの CU は、1 つ以上の処理エレメント (PE) とローカルメモリーで構成されます。work-group は単一の CU で実行されますが、work-item は 1 つ以上の PE で実行されることがあります。一般に、CU は SIMD 形式で work-item の小さなセット (*sub-group* として定義) を実行します。sub-group は NVIDIA では ワープ (warp) 、AMD ではウェーブフロント (wavefront) と呼ばれます。sub-group サイズは NVIDIA 向けには 32 で、AMD 向けには通常 64 (一部のアーキテクチャー向けには 32) です。

**計算**

カーネルを構成する work-group は、CU 全体にスケジュールされます。この時点で、それぞれの CU は処理エレメントで 1 つ以上の *sub-group* を実行します。計算ユニットには、算術演算を実行する整数論理ユニットや浮動小数点ユニット、メモリー操作を行うロード/ストアユニット、超越関数 (正弦、余弦、逆数、平方根など) を実行する特別なユニット、AI で役立つ行列操作など、さまざまな種類の処理エレメントが含まれます。処理エレメントが操作を完了するのに要する時間 (クロックサイクルで測定) は、レイテンシーと呼ばれます。レイテンシーは操作の種類によって異なります。例えば、グローバル・メモリー・トランザクションのレイテンシーは、レジスター呼び出しに比べ桁違いに大きく、これは各種算術演算でも同じことが当てはまります。

スループットは、実行された操作の数と、それらの完了に要する時間の比率です。この比率は、命令のレイテンシーを減らすか、同時に実行する命令数を増やすことで高めることができます。これまで、CPU はクロック周波数を上げて命令レイテンシーを最小化することでスループットを向上させてきました。一方、GPU はレイテンシーを隠匿することでスループットを向上させます。これにより、CU は sub-group 間で「コンテキスト」(レジスター、命令カウンターなど) をわずかな労力で変更できます。そのため、操作に多くのクロックサイクルを要する場合、CU は「コンテキスト」を変更し、別の *sub-group* の操作を実行することでそれらを隠匿できます。アーキテクチャーによって、同時に実行できる *sub-group* の最大数は異なります。実際に実行中の *sub-group* と実行中の *sub-group* の最大数の比率は「占有率」として定義されます。次の節で詳しく説明します。

GPU における work-item の同時実行は、複数レベルで実現されます。

1. 同一 *sub-group* 内の異なる *work-item* は SIMD 形式で同期実行されます。つまり、同じ操作が異なるデータを実行します。
2. 前述したように、CU はレイテンシーを隠匿するため、同一または異なる *work-group* から複数の *sub-group* を同時に実行します。
3. GPU を構成する CU は、異なる *work-group* に属する、異なる *sub-group* を同時に実行します。

これらの並列実行の機能は、起動されたカーネルが GPU 全体をビジー状態にする十分な大きさの *work-item* を持っている場合にフル活用されます。

**メモリー**

次の図は、ディスクリート GPU を搭載したシステムにおける一般的な接続方法を示しています。[1] ホストとデバイスを接続し、[2] CU をグローバルメモリーに接続します。例えば、NVIDIA* GA100 GPU の目安となる帯域幅は次のようになります。[1] PCIe* x16 4.0 では 31GB/秒、および [2] HBM2 では 1555GB/秒。

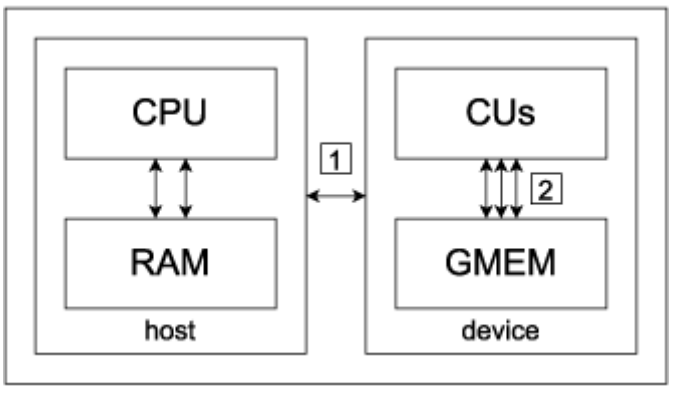


**図 1**

CPU と GPU 間の接続 [1] が大きなボトルネックになる可能性があります。そのため、ホストとデバイス間のデータ転送を慎重に検討し、GPU 上のデータの局所性を可能な限り維持することが重要です。ただし、カーネルの実行とオーバーラップすることで、PCIe* メモリーのトランザクションで生じるレイテンシーを隠匿することができます。

GPU の主要な特徴として、CU とグローバルメモリー間の高い帯域幅があります [3]。これは、それらを接続するメモリー・コントローラーの数と幅によるものです。例えば、NVIDIA* GA100 GPU には、12 個の 512 ビットの HBM メモリー・コントローラーがあります。これにより、クロックサイクルごとに大量のデータを転送できます。NVIDIA* GA100 GPU では、クロックごとに 6144 ビットです。ただし、この高帯域幅のメモリーを十分に活用するには、メモリーアクセスを結合する必要があります。つまり、work-item はキャッシュに最適な方法でメモリーアクセスする必要があります。

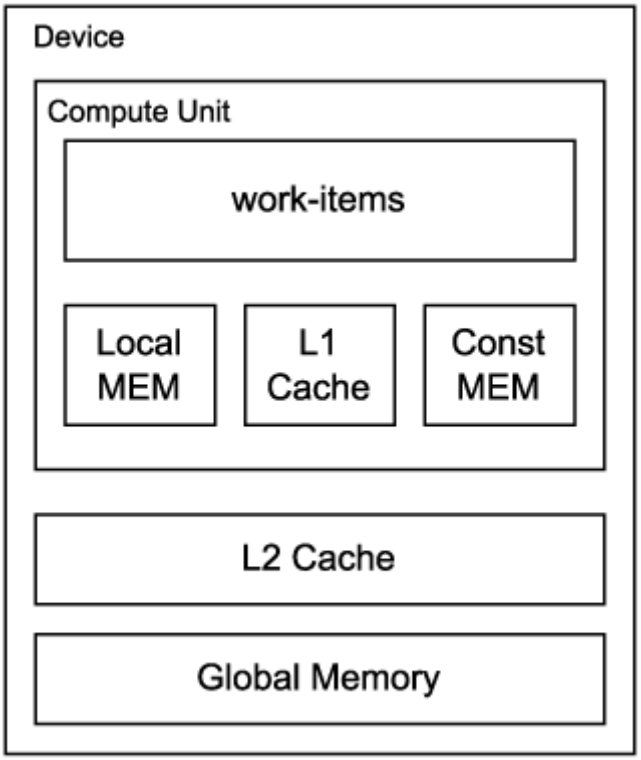


**図 2**

work-item とグローバルメモリー間にはいくつかのメモリー階層があります。以下に、それらをアクセス・レイテンシーが低い順に示します。

- **レジスター**は、ワークメモリーとして使用される work-item からは透過なデータを維持します。
- **コンスタント・メモリー**は、CU が使用する読み取り専用メモリーです。
- **ローカルメモリー**は CU ごとにあり、同一 work-group 内の work-item 間で共有されます。ローカルメモリーは、グローバルメモリーよりも高速であり、再利用されるグローバルメモリーのデータをキャッシュするために使用されます。
- **グローバルメモリー** (DDR または HBM) と CU を接続するメモリーシステムを構成する **L1** および **L2** キャッシュ。

## 最適化の目的

### 優先度

GPU コードのパフォーマンスに影響する主な要因を重要度の高い順に示します。

- **合成されていない**グローバル・メモリー・アクセス。キャッシュが完全に活用されると、メモリーアクセスは結合され、高い帯域幅を維持できます。結合の方法はアーキテクチャーによって異なりますが、一般に、同じ sub-group 内の work-item が連続したメモリー位置をアクセスすることで実現されます。
- ローカルメモリーの**バンク競合**。ローカルメモリーは複数のバンクに分割されており、異なる work-item から同時にアクセスできます。異なる work-item が同じメモリーバンクにアクセスすると、バンク競合が発生してトランザクションはシリアル化されます。
- `if` 文などの条件式やループの反復回数は work-item によって異なるため、同じ *sub-group* に属する work-item が異なる命令を実行することで**発散**が発生します。近年のアーキテクチャーではこの事象が緩和され、パフォーマンスのペナルティーが軽減されています。

計算の種類が異なれば最適化の優先順位も変わってきます。例えば、メモリー・トランザクションに対し算術演算が少ないメモリー依存のタスクを考えてみます。この場合、GPU を十分に活用するには、メモリーアクセスを結合することが重要です。一方、メモリー・トランザクションに対し算術演算が多い計算依存タスクがあります。この場合、スレッドの発散を回避することが有用な場合があります。算術演算数とリード/ライトデータのバイト数の比率は、**演算強度**として定義されます。

`I = (浮動小数点操作数) / (リード/ライトデータのバイト数) [FLOP/バイト]`

**ルーフライン・モデル**を利用して、カーネルの演算強度をハードウェア特性に関連付けることで、カーネルがメモリー依存であるか計算依存であるか確認できます。**ルーフライン・モデル**は 2 次元座標として表示され、x 軸には演算強度が、y 軸には浮動小数点演算のスループット (FLOPS: 1 秒あたりの浮動小数点演算) が示されます。

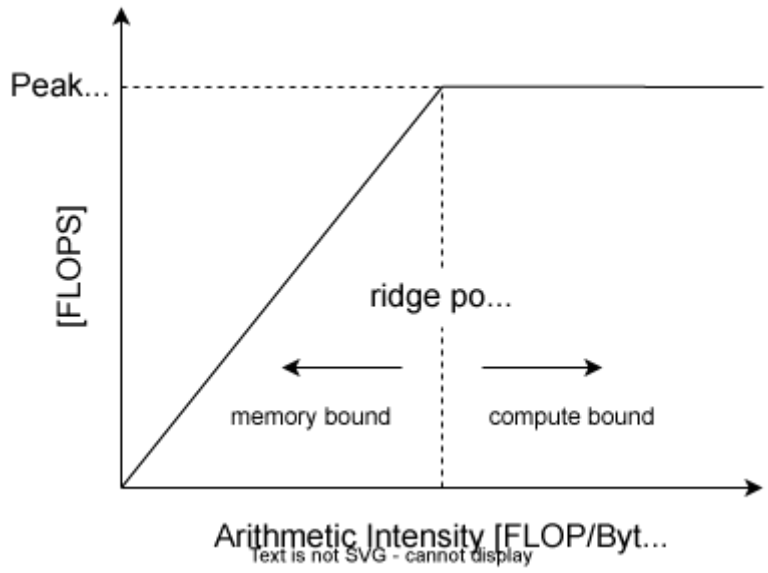


**図 3**

実際の**ルーフライン**を構成する最初のセグメントは、`y = x * B` を示します。ここで、`B` はグローバル・メモリー・システムの帯域幅です。次に水平線 (`y = Pmax`) は、FMA など特定の操作の最大浮動小数点スループット (`Pmax`) に依存します。セグメントが遭遇するポイントは**リッジポイント**と呼ばれます。

カーネルのパフォーマンスは、**ルーフライン**にプロットされたポイント (点) で示されます。x 軸はカーネルの演算強度を示し、y 軸は計測されたカーネルの FLOPS を示します。このポイントがリッジポイントの左にある場合、そのカーネルは**メモリー依存**であり、右にある場合は**計算依存**です。

**占有率**

カーネルのパフォーマンスを評価するには、その**占有率**を考慮します。占有率は、次のように定義される計算ユニットの式で求められます。

```
占有率 = アクティブな sub-group 数 / アクティブな sub-group の最大数
```

アクティブな sub-group は、CU で実際に実行される sub-group です。アクティブな sub-group の最大数は、計算ユニットのアーキテクチャーによって異なります。例えば、NVIDIA* GA100 CU アーキテクチャーでは 64 です。

占有率を高めるにはアクティブな sub-group 数を最大化する必要があります。ただし、計算ユニットのアーキテクチャーによって制約は異なります。

- **work-group あたりの work-item の最大数**
- **CU で実行される work-group の最大数:** work-group サイズが小さすぎると、CU はアクティブな sub-group の最大数を実行することができません。
- **レジスター数の制限:** カーネルコードが複雑になるとレジスターの使用量が増加します。コードを簡素にすることでレジスターの使用量を軽減できます。これは、コードを複数のカーネルに分割することで実現することもできます。
- **ローカルメモリー量の制限:** work-group がローカルメモリーを消費しすぎると、同時に実行できる work-group 数が減少します。

work-group が使用するレジスターやローカルメモリーが多いと、占有率が制限される可能性があります。ユーザーは、work-group のサイズを変更することで占有率を改善できます。このサイズは、sub-group サイズの倍数で、アクティブな sub-group の最大数の除数である必要があります。

例えば、NVIDIA* GA100 GPU では、各 work-item は最大 32 個のレジスターを使用して完全な占有を実現できます。

```
r_max = (CU ごとのレジスター数) / (アクティブな sub-group の最大数) * (sub-group サイズ) = 32
```

work-item が 32 未満のレジスターを使用する場合、CU で同時に実行できる work-group の最大数 `wg_max` とすると、ローカルメモリーにも同じことが当てはまります。

```
wg_max = (アクティブな sub-group の最大数) * (sub-group サイズ) / (実際の work-group サイズ)
```

各 work-group が最大 `48Kb / wg_max` のローカルメモリーを割り当てる場合、完全な占有率が得られます。

実効占有率は重要ですが、パフォーマンスにおける最重要のメトリックではありません。命令レベルの並列処理が十分にあり、同じ *sub-group* に属する独立した命令の同時実行が可能であれば、低い占有率でもレイテンシーを十分に隠匿できます。これについては、こちら (英語) をご覧ください。

さらに、GPU のすべての CU を利用するためカーネルで起動される work-item の最小数は、少なくとも次の `wi_min` でなければなりません。

```
wi_min = (アクティブな sub-group の最大数) * (sub-group サイズ) * (CU 数) = 262144
```

これらのパラメーターはすべて、特定ベンダーの各アーキテクチャーのドキュメントに記載されていますが、以下の表にいくつかの一般的な GPU アーキテクチャーの数値を示します。

一般的な GPU アーキテクチャーのリファレンス占有率

| アーキテクチャー | アクティブな sub-group の最大数 | work-item の最大数 | work-group の最大数 | レジスター数 | ローカルメモリー (バイト) |
|---|---|---|---|---|---|
| NVIDIA* S.M. 7.0 | 64 | 2048 | 32 | 65536 | 65536 |
| NVIDIA* S.M. 7.5 | 32 | 1024 | 16 | 65536 | 65536 |
| NVIDIA* S.M. 8.0 | 64 | 2048 | 32 | 65536 | 65536 |
| AMD* GFX9xx | 40[1] | 1024 | 16 | 29184 (?) | 65536 |

**[1]** この図は、AMD アーキテクチャー全般に適用されます。work-group が 1 つの sub-group (例えば 64 work-item) のみである場合、CU あたりの work-group の最大数は 40 です。

NVIDIA* GPU の場合、NVIDIA* Nsight* Compute (英語) は占有計算機を提供しており、理論上の占有率がどのように計算されるか判断するのに役立ちます。非推奨ですが、NVIDIA* の spreadsheet (英語) でも同様の機能を提供することを示しています。

## パフォーマンス解析

パフォーマンス解析と最適化は繰り返し作業です。開発者は、ツールを使用してアプリケーションのパフォーマンスを測定しボトルネックを特定して、それらを改善しながら、この手順を繰り返します。それぞれの反復作業で、以前は見つからなかったボトルネックが明らかになることがあります。

ある時点で、アプリケーションの制限要因となる部分で、可能な限り高いパフォーマンスを特定することが重要です。これは、光速またはルーフラインと呼ばれることもあり、アプリケーションの理論上のピーク・パフォーマンスを予測したり、そのパフォーマンスにどれだけ近づいているかを判断するのに役立ちます。

以降の節では、解析ツールと制限要因について詳しく説明します。

### 解析の方法論

パフォーマンス解析に使用されるツールはプロファイラーとも呼ばれます。プロファイルという用語はいろいろな意味で使用されます。ここでは、パフォーマンス解析に使用されるツールの総称という意味で使用します。特定のパフォーマンス・ツールの説明では、より具体的な意味で使用されることがあります。

パフォーマンス解析は、大きく分けてトレースとサンプリングに分類されます。トレースは、アプリケーションの実行中に 1 つ以上のイベントが発生するたびに記録します。サンプリングは、実行中のアプリケーションの状態を定期的に調査して、その状態を記録します。頻繁に発生するイベントでは、トレースで大量のデータが蓄積される可能性があります。サンプリングでは、サンプリング間隔を調整することでデータ量を制御できます。間隔を長くするとデータ量は減りますが、短い間隔の動作を記録できないことがあります。どちらも改良すべき点はありますが、トレースまたはサンプリングのいずれかを実行中のデータ軽減と組み合わせることができます。

どの解析ツールでも、考慮すべき 2 つのことがあります。

- **オーバーヘッド:** ツールが通常のプログラムの実行時間をどれくらい増加させるかを表わします。パフォーマンス・ツールは、オーバーヘッドを最小限に抑えることが求められます。ただし、オーバーヘッドの増加を十分に理解している場合は、データを解釈する際にこれを補正できます。例えば、あるツールはコードの GPU 実行領域では正確な結果を提供し、CPU 実行領域では実行時間が長くなります。
- **データ量:** 生成されるファイルの大きさを示します。データ量が多いと、オーバーヘッドも増加します。また、大きな出力データセットは管理が困難で、特に出力データセットを表示するためリモートマシンに移動する場合、後処理ツールの応答性にも問題があります。

### システムレベルの解析

システムレベルの解析では、同一ノードまたは異なるノード上のプロセス間の相互作用、および CPU と GPU 間の相互作用を調査します。

複雑なワークロードにおける CPU と GPU 間の相互作用を解析するのは困難なことがあります。ベンダーは、このような解析を支援するためトレースツールを提供することがあります。それらは、メモリー割り当て、メモリー転送、カーネルの起動、同期など、GPU 間の API 呼び出しのタイムスタンプと期間を記録します。これらのツールには、シリアル化や過度のアイドル時間などのボトルネックを視覚的に特定するタイムライン表示が含まれます。

状況に応じて、OS のカーネルトレース (Linux* ftrace など) を使用して、それをアプリケーションの実行に関連付けると便利です。これには、root 権限が必要になります。パフォーマンスの問題に関連するカーネルのアクティビティーが理解できない場合は、循環バッファーを利用するすべての OS アクティビティーを記録し、パフォーマンスの問題が検出されたときにアプリケーションの制御下でバッファーをダンプすると便利です (例えば、タイムステップが平均時間や予測時間よりも大幅に長くかかる場合)。循環バッファーによる手法は、トレース・データ・ストリーム全体を記録する際にコストが高い場合に有効です。

分散アプリケーションのスケーリング (通常メッセージ・パッシング・インターフェイス (英語) を使用) は特筆に値します。一般に使用されるスケーリングには 2 つの定義があります。**強力なスケーリング**は、問題のサイズを一定に保ち、MPI ランクの数が増加するのにしたがって経過時間を測定します。**弱いスケーリング**は、MPI ランクの数に比例して問題サイズを大きくします。

強力なスケーリングはより困難な問題です。多くの場合、すべての MPI ランクをビジーに保つのに十分なワークがありません。MPI プロファイル・ツールを利用して、異なる数のランクでスイープを実行し、MPI プロファイルを比較することが有用です。

特に大規模なスケーリングでは、そのほかの MPI の問題がしばしば発生します。リダクション操作は `log(N)` に反比例します。さらに、小さなリダクション操作 (スカラー値への `MPI_Allreduce` など) は、OS によるノイズの影響を受ける可能性があります。ネットワークが混雑する可能性があるため、大規模な共有クラスターではポイントツーポイント操作でも影響を受ける可能性があります。強力なスケーリングでは、メッセージサイズは通常、ランク数が多いほど小さくなるため、MPI レイテンシーがさらに重要になります。

開発者は、アプリケーションの動作が大規模なノードと小規模なノードで実行される際の違いを予測する必要があります。通常のように MPI プロファイル・ツールを使用すると、動作の違いを理解するのに役立ちます。オーバーヘッドの低いツールは、大規模なケースでは特に重要です。

**カーネルレベルの解析**

カーネルレベルの解析では、GPU カーネルの実行に費やされた時間と、個々の GPU カーネルのパフォーマンスに注目します。

前の節で説明したツールは、通常、起動パラメーター、起動回数、カーネルで消費された時間など、カーネル実行ごとのサマリーを示します。アプリケーションの合計時間は、CPU の経過時間と GPU の経過時間の合計として見積もられることが多く、GPU での経過時間はカーネルの実行時間の合計として概算されます。これにより、GPU カーネルの実行時間を改善することで、全体でどれだけ改善されるかが分かります。実行がオーバーラップしていたり、データの転送時間が長い場合は、常に正確であるとは限りませんが、経験則としては適切です。

カーネルのパフォーマンスを詳しく解析するには以下が必要です。

- カーネルのソースコードを調査
- カーネル向けにコンパイラーが生成するアセンブリー言語の調査
- カーネル実行中のハードウェア・パフォーマンス・メトリックの収集

アセンブリー言語を生成する方法は、コンパイラーと GPU によって異なります。詳細については以降で説明します。

ここからは、GPU (多くの場合 CPU にも該当) で利用可能なメトリックと、それらを解釈してパフォーマンスを改善する作業で導入できる一般的な手法について説明します。異なる GPU 向けの詳細については、このドキュメントの後半で説明します。

**重要な GPU メトリック**

**レートメトリック**

アプリケーションが GPU を使用するのは、利用可能な計算リソースを増やすためです。通常、計算スループットは、単位時間あたりに処理される演算数で示されます。例えば、倍精度浮動小数点演算の数/秒、32 ビット整数演算の数/秒などです。特定の GPU では、これらのピーク値が公開されています。

多くの場合、アプリケーションのピーク・パフォーマンスは、非計算リソース (特にメインメモリーやスクラッチパッド・メモリーなどさまざまなメモリー領域) へのアクセスによって制限されます。ここにもピーク値があります。例えば、メインメモリーの帯域幅は、単位時間あたりのバイト数で表現されます。

従来のルーフラインのようなモデルでは、ほかのリソースによる制限 (一般的なものはメインメモリーの帯域幅) を考慮して、達成可能な計算パフォーマンスを定量化しようとします。アプリケーションが計算以外のメトリックでピーク・パフォーマンスに達している場合、ピーク計算パフォーマンスを達成することはできません。これにより、開発者は、アプリケーションで達成可能なピーク・パフォーマンスに関する情報を得ることができます。

**利用率メトリック**

特定のリソースや機能ユニットがどれだけビジーであるかを知るのは有用です。この利用率メトリックは、レートメトリックとは異なります。リソースの利用率が高くても、ピーク・パフォーマンスにほど遠い場合があります。1 つの例として、メモリー・アクセス・パターンが不均一なカーネルが挙げられます。この場合、メモリー帯域幅がピークから離れていても、メモリーユニットの利用率は非常に高くなることがあります。利用率メトリックは、ルーフライン・モデルでは明らかにならないボトルネックを理解するのに役立ちます。

通常、利用率メトリックはメモリーユニットと計算ユニットで利用できます。また、キャッシュやローカルメモリーなど、各種マイクロアーキテクチャー・ブロックでも利用できることがあります。

**発散**

前述のように、GPU は複数の work-item を SIMD (単一命令複数データ) 方式で同時に実行する複数の計算ユニット (CU) で構成されています。

開発者は、単一の work-item に対して実行する操作を記述します。コンパイラーは、このコードを複数の work-item を同時に処理する命令に変換します。各 GPU には、sub-group サイズと呼ばれる、同時に実行される work-item の最小数がネイティブに設定されています。

発散 (Divergence) は、異なる work-item が異なるパスをたどることで発生します。多くの work-item が特定の命令で実行される場合、コンパイラーは可能なすべてのパスの組み合わせを考慮して命令を生成する必要があります。特定の命令で非アクティブな work-item は無効になります。これにより SIMD レーンの一部しか利用されないため、利用率は低下します。

GPU は発散を測定するメトリックを提供し (通常、sub-group ごとにアクティブな work-item)、ネイティブの sub-group サイズと比較できます。

**占有率**

GPU の占有率については前述しましたが、これは簡単に言うと、特定のカーネルで実際にアクティブな sub-group の数と、アクティブな sub-group の理論上の最大数との比率です。占有率は、カーネルが利用可能な最大の並列性をどれくらい活用できているかを開発者に示すため重要です。

一部の GPU には、実際の占有率を測定するハードウェア機能が備わっています。理論上の占有率は、コンパイルされたカーネルとハードウェアのプロパティーから計算できます。

### 起動パラメーター

カーネルは、グローバルレンジとローカルレンジで起動されます。後者は work-group のサイズです。work-group のサイズは、sub-group サイズの倍数である必要があります。そのため、グローバル問題サイズを切り上げたり、グローバル問題サイズ外の work-item を処理しないようカーネルにコードを追加する必要があります。

占有率を改善するため、特定の GPU ハードウェアの work-group サイズに制約が課される場合があります。CU 数など、特定の GPU ハードウェアと何らかの関連性のあるグローバル問題サイズを選択することも有益な場合があります。グローバルとローカルの問題サイズは自然なサイズに合わせる必要がなく、ハードウェアに適合するように選択できます。

すべての GPU は、カーネルの起動ごとに実際の起動パラメーターを確認するメカニズムを提供しています。これには、グローバルおよびローカル問題サイズ、レジスター数、およびローカル・メモリー・サイズなどのカーネル・プロパティーが含まれます。

## NVIDIA* GPU 上のパフォーマンス

このセクションでは、SYCL* プログラミング・モデルのコンテキスト内で NVIDIA* GPU アーキテクチャーと関連するパフォーマンスの考慮事項を説明します。NVIDIA* アーキテクチャー固有のパフォーマンスに関する最新の考慮事項については、NVIDIA* のドキュメント (英語) を参照してください。

### アーキテクチャー

**注意:** ここでは、CUDA* と SYCL* 用語の簡単な対比を示しています。詳しい説明については、ComputeCpp の『SYCL* for CUDA* Developer』(英語) を参照してください。ただし、ComputeCpp のガイドでは SYCL* 1.2 のみをカバーしているため、SYCL* 2020 で導入された USM など、いくつかの重要な機能については説明されていません。

NVIDIA* GPU の基本計算ユニットは、ストリーミング・マルチプロセッサーまたは SM と呼ばれます。SM は、32 個の work-item で構成される sub-group を実行します。NVIDIA では sub-group を ワープ (*warp*) と呼んでいます。work-item を実行するエンティティーはスレッド (*thread*) と呼ばれます。

work-group は、スレッドブロック (*thread block*) または協調スレッドアレイ (CTA: *cooperative thread array*) と呼ばれます。これには通常の規則が適用されます。CTA は同じ SM で同時に実行されることが保証され、SYCL* ローカルメモリー (CUDA* では共有メモリー) などのローカルリソースを利用でき、work-item 間で同期できます。

以下は、NVIDIA*/CUDA* と SYCL* の用語の対比表です。

| NVIDIA*/CUDA* | SYCL* |
|---|---|
| ストリーミング・マルチプロセッサー (SM) | 計算ユニット (CU) |
| ワープ | sub-group |
| スレッド | work-item |
| スレッドブロック | work-group |
| 協調スレッドアレイ (CTA) | work-group |
| 共有メモリー | ローカルメモリー |
| グローバルメモリー | グローバルメモリー |
| ローカルメモリー | プライベート・メモリー |

work-group サイズは、sub-group サイズの 32 の倍数である必要があります。適切な work-group サイズは、占有率を最大化するように選択され、カーネルによって使用されるリソースに依存します。これは 1024 を越えることはありません。

NVIDIA* GPU では、デバイスクエリー `sycl::info::device::sub_group_sizes` が、値 32 の単一要素を持つベクトルを返します。

work-group サイズを選択したら、グローバルサイズを選択します。グローバルサイズを計算ユニット数の倍数にすると、負荷分散に役立つことがあります。デバイスクエリー `sycl::info::device::max_compute_units` は、計算ユニットの数を返します。

次に、それぞれの work-item (カーネル・インスタンス) が問題の複数の項目で動作するようにカーネルコードを記述します。起動パラメーターは、問題サイズにかかわりなく、ハードウェアのレイアウトに基づいて調整できます。

例えば、次の SYCL* コードは、ハードウェア・レイアウトに基づいて起動パラメーターを決定し、カーネル内のループで問題サイズを調整します。CUDA* では、このタイプの内部ループはグリッドストライド (*grid-stride*) と呼ばれます。

```
int N = some_big_number;
int wgsize = 256;
int ncus = dev.get_info<info::device::max_compute_units>();
int nglobal = 32 * ncus;
cgh.parallel_for(nd_range<1>(nglobal * wgsize, wgsize),
  [=](nd_item<1> item)
  {
    int global_size = item.get_global_range()[0];
    for (int i = item.get_global_id(0); i < N; i += global_size)
      y[i] = a * x[i] + y[i];
  });
```

### SM

NVIDIA* SM は、SM サブパーティション (英語) と呼ばれる 4 つの処理ブロックに分割されます。それぞれのワープは、存続期間全体で単一サブ・パーティションに存在します。ワープが停止すると、ハードウェア・ワープ・スケジューラーは準備ができている別のワープにスイッチできます。これは、いつでも実行できる十分な数のワープが必要であることを意味します。以降で、ハードウェア・メトリックを使用して、ワープ・スケジューラーの効率とストール時間を評価する方法を紹介します。

### メモリー

メモリーにはいくつかの種類があります。

- **SYCL* グローバルメモリー** (NVIDIA* グローバルメモリー) はデバイス上にありますが、一部のグローバルアドレスは、ホストまたは別のデバイス上にある CUDA* 管理メモリー (SYCL* USM を使用) を参照する場合があります。すべてのグローバル・メモリー・アクセスは、すべての CU で共有される L2 キャッシュを経由します。カーネルの存続期間中のみ読み取られるデータは、`sycl::ext::oneapi::experimental::cuda::ldg` 関数を使用して CU ごとの L1 キャッシュに配置することができます。
- **SYCL* プライベート・メモリー** (NVIDIA* ローカルメモリー) は、特定の work-item からのみアクセスできますが、グローバルメモリーにマップされているため、グローバルメモリーと比較してパフォーマンス上の優位性はありません。work-item ごとに同じアドレスにマップされます。これは、work-item スタック、レジスタースピル、およびその他の work-item のローカルデータに使用されます。
- **SYCL* ローカルメモリー** (NVIDIA* 共有メモリー) は work-group で使用できます。グローバルメモリーよりも帯域幅が広く、レイテンシーが短縮されます。SYCL* ローカルメモリーには 32 のバンクがあります。連続する 32 ビット・ワードは、異なるバンクに割り当てられます。
- **SYCL* 共有メモリー** (NVIDA* 管理メモリー) は、ホストまたはアクセスされる任意のデバイス (上記のグローバルメモリーの説明を参照 ) に配置できます。ホストとデバイス間、またはデバイスとデバイス間でのメモリーの移行は、ランタイムによって管理されます。ただし、ユーザーは SYCL* 2020 API の `prefetch` や `mem_advise` を介して動作を制御するパフォーマンスのヒントを指定できます。

### バンク競合

SYCL* ローカルメモリー内の 32 バイト float 配列にアクセスするストライド 1 のループを考えてみてください。最初の sub-group では、work-item 0 は配列要素 0 にアクセスし、work-item 1 は配列要素 1 にアクセスします。つまり、sub-group 全体として配列要素 0:31 にアクセスするため、バンク競合は起こりません。

一方、ループのストライドが 32 である場合、work-item は要素 32、64、96、... にアクセスし、sub-group は 32 要素離れた 32 の位置にアクセスするため、結果として 32 ウェイのバンク競合が発生します。これは、SYCL* ローカルメモリーの帯域幅を 1/32 に減少させます。

Volta のプレゼンテーション (英語) では、スライド 54 から 72 で共有メモリーバンクの競合について分かりやすく説明しています。

バンク競合を測定するハードウェア・メトリックがあります。

### ローカル・メモリー・サイズ

デフォルトでは、カーネルはスレッドブロックあたり 48KB の SYCL* ローカルメモリーに制限されます。より大きな割り当てをサポートするプラットフォーム (Compute Capability 7.0 以降) では、`SYCL_PI_CUDA_MAX_LOCAL_MEM_SIZE` 環境変数を割り当て最大バイト数に設定して、明示的に有効にする必要があります。`sycl::info::device::local_mem_size` は、現在の SYCL* ローカルメモリーの最大値を返します。

Compute Capability 7.0 以降では、SYCL* ローカルメモリーと L1 キャッシュは、物理的に同じメモリーダイ上に配置されます。つまり、それらの合計は固定値であることを意味します。そのため、大量の共有メモリーを割り当てると、利用可能な L1 キャッシュが縮小してパフォーマンスに影響する可能性があります。

**警告:** Compute Capability 7.0 以降のカードでは、前述のように L1 キャッシュとローカルメモリーはハードウェアによって結合されています。共有メモリーはプラットフォーム機能によって任意のサイズに設定できますが、L1 にはそのような柔軟性はなく、固定値 (カーブアウトと呼ばれます) しか利用できません。そのため、ローカルメモリーを少し多く割り当てると、ドライバーは L1 を制限する次のカーブアウト値に切り替える可能性があります。

前述のように、メモリータイプ間の分割は実行時にドライバーによって行われます。`sycl::local_accessor` を使用して、割り当てられた実際のメモリー量に基づいて決定します。そのため、`SYCL_PI_CUDA_MAX_LOCAL_MEM_SIZE` を過度に大きく設定しても、選択した L1 キャッシュの量に影響することはありません。

特定の Compute Capability に対するこれらの適正値は、CUDA* C プログラミング・ガイド (英語) で確認できます。NVIDIA* では、「L1 キャッシュ」と「テクスチャー・キャッシュ」という用語が同じ意味で使用されることに注意が必要です。

### メモリー結合

異なる work-item は通常、グローバルメモリー内の異なる位置にアクセスします。同じロード命令内の特定の sub-group でアクセスされるアドレスが、キャッシュラインの同じセットである場合、メモリーシステムは最小数のグローバルメモリーへのアクセスを発行します。これはメモリー結合と呼ばれます。この要件は、隣接するメモリー位置にアクセスする work-item によって容易に満たすことができます。データ構造を 32 バイト境界に配置することで、パフォーマンスをさらに向上できます。間接アクセス/大きなストライドによって、明らかにこれをが困難であることがあります。

最近の NVIDIA* GPU では、洗練されたキャッシュとメモリーシステムを備えているため、グローバル・メモリー・アクセスが結合される可能性は高くなります。結合の測定に有効なハードウェア・メトリックがあります。

### キャッシュ

すべての GPU ユニットは L2 キャッシュを共有します。L2 キャッシュは物理アドレスでアクセスされます。これには、データ圧縮とグローバルアトミック (浮動小数点加算など) の機能も含まれます。

各 SM には、複数の機能に使用される L1 キャッシュがあります。L1 のスループットは、パフォーマンスを制限する要因になることがあります。

### 占有率

NVIDIA* では、同時にアクティブな CU 数を、利用可能な最大 CU 数で割ったものを占有率として定義しています。明らかに、占有率が上がると GPU 利用率が上がり、パフォーマンスが向上することが期待されます。理論上の占有率は、ハードウェアの制限、カーネルで使用されるレジスター数、およびカーネルで使用される共有メモリーの量によって決定されます。

理論上の占有率は、work-group サイズと特定のカーネルを起動するパラメーターを決定するガイドラインとなります。NVIDIA* プロファイリング・ツールは、起動されるカーネルごとの実際の占有率と理論上の占有率を示します。実行時に work-group サイズを変更して、結果をベンチマークして最適な work-group サイズを選択できます。しかし、コードを実行することなく理論的な占有率の推測を利用することもできます。

NVIDIA は、オンラインの spreadsheet (英語) を提供しています。公式には非推奨ですが、NVIDIA* Nsight* Compute の占有率セクションと同じ機能を提供し、理論上の占有率がどのように決定されるかを理解するのに役立ちます。代わりに NVIDIA* Nsight* Compute (英語) を使用することが推奨されています。

### パフォーマンス・ツール

一般に、NVIDIA* が CUDA* 向けに提供するすべてのパフォーマンス・ツールは、DPC++ CUDA* プラグインを使用して SYCL* アプリケーションをシームレスに処理します。

この節では、それらのパフォーマンス・ツールの一部を紹介します。

### NVIDIA Nsight* Systems (nsys)

NVIDIA Nsight* Systems (英語) は、コマンドライン・ツール *nsys* と GUI *nsys-ui* を含んでいます。さらに、NVTX (英語) トレース・ライブラリーをアプリケーションで使用して、nsys 解析の対象領域を制限し、アプリケーション固有のイベントを nsys 解析に追加することができます。

### 基本的な使い方

```
$ nsys profile <command> <arguments>
```

このコマンドは、指定するコマンドを与える引数で実行し、デフォルト設定でプロファイルを行います。出力は、NVIDIA* 固有の形式でレポートファイルに書き込まれます。

nsys はシステム全体を監視することに注意してください。コマンドは何であっても構いませんが、nsys は単にコマンドを起動して監視を開始し、コマンドが終了するまで監視を続行します。

```
$ nsys stats <report file>
```

このコマンドは、レポートファイルから sqlite データベースを作成し、データベースでいくつかのレポートを作成します。レポートには、API の使用状況、GPU に関連するメモリーコピー、および GPU カーネルのタイミングなどが含まれます。結果はデフォルトでコンソールに出力されます。レポートを csv 形式のファイルに書き込むオプションもあります。

nsys-ui コマンドを使用して、レポートファイルを GUI で調査することもできます。これには、ローカルまたは VNC のようなリモート X ビューアーを介して接続されたディスプレイが必要です。レポートファイルは、ラップトップなどのローカルマシンに移動して、そこで GUI を実行できます。レポートファイルは非常に大きくなる可能性があり、大量のメモリーを必要とする場合があります。

アプリケーションで NVTX アノテーションを戦略的に使用し、コマンドライン引数を追加することで、レポートサイズを縮小し、解析ワークフローを円滑にすることができます。

レポートは JSON 形式でエクスポートすることもできます。これには、sqlite データベースと同じデータが含まれていますが、SQL を熟知していない開発者にも解析が容易です。

**NVTX アノテーション**

NVTX は、NVIDIA* が提供するインストルメント API で、パフォーマンス・ツールで利用されます。この API は、`nvToolsExt.h` をインクルードすることで利用できます。ライブラリーのリンクなどは必要ありません。

NVTX は、大規模なアプリケーションで、ほかの NVIDIA* ツールのデータ収集をアプリケーションの特定の部分に制限するのに役立ちます。

NVTX は、デバイスコードではなく、ホストコードの測定にのみ利用できることに注意してください。

NVTX の使い方と機能の詳細については、NVIDIA* のドキュメント (英語) を参照してください。

**NVIDIA* Nsight* Compute (ncu)**

前の節では、NVIDIA Nsight* Systems (nsys) について説明しました。NVIDIA* Nsight* Compute (ncu) は、GPU ハードウェアのパフォーマンスに注目する支援ツールです。ncu を使用すると、NVIDIA* GPU で利用可能なハードウェア・カウンターにアクセスできるだけでなく、特定のカーネルが GPU をどの程度活用しているかを理解するのに役立つ事前定義された解析タイプ (セクションと呼ばれます) を知ることができます。

ncu には GUI 版がありますが、ここでは nsys の説明のようにコマンドラインといくつかの処理スクリプトを利用します。ncu CLI については、https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html (英語) で説明されています。

**ncu オーバーヘッド**

ncu は、アプリケーションの実行時間を大幅に増加させる可能性があります。いくつかの原因が考えられます。

- 指定されたメトリックを収集するため、場合によってはカーネルを複数回実行する必要があります。
- 一部のメトリックでは、オーバーヘッドが高いカーネルのバイナリー・インストルメンテーションが必要になります。
- 通常、カーネルの実行はシリアル化されるため、同時実行性が低下します。

詳細については、「カーネル・プロファイル・ガイド」(英語) を参照してください。

速度が低下すると、通常、収集プロセスを制限する必要があります。NVIDIA* では、これに対処するいくつかの方法を提案しています。

- nsys の節で説明したように、NVTX を使用してアプリケーションをインストルメントし、ncu にコマンドライン・オプション `--nvtx` および `--nvtx-include` または `--nvtx-exclude` を指定して、特定の NVTX レンジを含めるか除外するかを指示します。ドメイン内のレンジを指定する構文は、nsys とは逆であることに注意してください。`range@domain` ではなく `domain@range` となります。ncu の構文は非常に豊富ですが複雑です。「NVTX フィルター処理」(英語) のセクションを参照してください。
- `-k kernelname` を指定して、単一カーネルのデータのみを収集します。カーネル名には正規表現を利用できます。

  **注意:** C++ ユーザーは、`--kernel-name-base=mangled` を指定してマングルされた名前を使用することもできます。

- `--launch-count` と `--launch-skip` を使用して、特定数のカーネルの起動を収集します。

上記は組み合わせて使用できます。

**セクション**

ncu には、セクションと呼ばれる事前定義された多数のメトリックのセットが用意されています。各セクションは、特定のパフォーマンスに関する疑問を解決するのを支援します (例: アプリケーションはメモリー依存であるか、など)。

セクションは、`ncu --list-sections` で一覧を表示できます。ncu バージョン 2022.1.1.0 の 出力を以下に示します。

| 識別子 | 表記名 |
|---|---|
| ComputeWorkloadAnalysis | Compute Workload Analysis |
| InstructionStats | Instruction Statistics |
| LaunchStats | Launch Statistics |
| MemoryWorkloadAnalysis | Memory Workload Analysis |
| MemoryWorkloadAnalysis_Chart | Memory Workload Analysis Chart |
| MemoryWorkloadAnalysis_Deprecated | (Deprecated) Memory Workload Analysis |
| MemoryWorkloadAnalysis_Tables | Memory Workload Analysis Tables |
| Nvlink | NVLink |
| Nvlink_Tables | NVLink Tables |
| Nvlink_Topology | NVLink Topology |
| Occupancy | Occupancy |
| SchedulerStats | Scheduler Statistics |

| 識別子 | 表記名 |
| --- | --- |
| SourceCounters | Source Counters |
| SpeedOfLight | GPU Speed Of Light Throughput |
| SpeedOfLight_HierarchicalDoubleRooflineChart | GPU Speed Of Light Hierarchical Roofline Chart (Double Precision) |
| SpeedOfLight_HierarchicalHalfRooflineChart | GPU Speed Of Light Hierarchical Roofline Chart (Half Precision) |
| SpeedOfLight_HierarchicalSingleRooflineChart | GPU Speed Of Light Hierarchical Roofline Chart (Single Precision) |
| SpeedOfLight_HierarchicalTensorRooflineChart | GPU Speed Of Light Hierarchical Roofline Chart (Tensor Core) |
| SpeedOfLight_RooflineChart | GPU Speed Of Light Roofline Chart |
| WarpStateStats | Warp State Statistics |

**メトリック**

セクションの代わりに、`--metrics` を使用して特定のメトリックを収集するよう ncu に指示できます。利用可能なメトリックは、`--query-metrics` で照会できます。

**出力**

この節の例では、解析が容易な出力を生成するため、`--csv` オプションを使用しています。`--log-file` オプションは、この出力と他の出力をともに `stdout` の代わりにファイルへ送ります。

プロファイル・レポートは、ncu GUI によって使用されます。これは、`--export` オプションで保存できます。その後、ファイルを別のマシンに移動して ncu-gui を使用し、ローカルで表示できます。

分断された名前は、`--print-kernel-base=mangled` で選択できます。

**カーネル・アセンブリーの抽出**

状況によっては、特定のカーネルのパフォーマンスを理解するため、アセンブリーを調べることが有効な場合があります。NVIDIA* の場合は、コンパイラーがカーネル向けに生成した PTX を調査します。

NVIDIA* GPU 用にビルドされた SYCL* アプリケーションから PTX を抽出するには、環境変数 `SYCL_DUMP_IMAGES` に `1` を設定してアプリケーションを実行します。これにより、現在の作業ディレクトリーに `sycl_nvptx641.bin` のような名前のファイルが生成されます。これらのファイルは CUDA* **fat バイナリー**であり、NVIDIA* ターゲットに依存しない仮想アセンブリー言語である PTX と、単一または複数のターゲットのマシンコードである SASS が含まれます。

次のように CUDA* ツール cuobjdump (英語) を使用して、FAT バイナリーから PTX と SASS の両方を抽出できます。

```
# Extract PTX
cuobjdump --dump-ptx sycl_nvptx641.bin

# Extract SASS
cuobjdump -sass sycl_nvptx641.bin
```

NVIDIA* Nsight* Compute ツールの GUI バ—ションでも逆アセンブリーを表示できます。

**モジュールの分割と読み込み時間**

コンパイルされたデバイスコードは、出力バイナリー (実行可能ファイルまたは共有ライブラリー) の「モジュール」と呼ばれる自己完結型のセクションに埋め込まれます。各モジュールは 1 つまたは複数のカーネルで構成され、PTX のみ、または PTX と SASS の両方を含む場合があります。プログラムがカーネルを実行する必要がある場合、モジュールをホストメモリーにロードし、対応するデバイスコードをデバイスにロードするようにドライバーに要求します。特定のデバイス・アーキテクチャーに適した SASS が使用できる場合は、直接ロードされます。それ以外の場合は、PTX から SASS へのジャストインタイム・コンパイルがトリガーされます。モジュールに複数のカーネルが含まれている場合は、カーネルの 1 つを初めて使用したときにモジュール全体がロードされます。

複数のカーネルを 1 つのモジュールにマージすると、プログラムの実行中にそのカーネルの大部分が使用される場合、一般にパフォーマンスが向上する可能性があります。これにより、バイナリーファイルからデバイスコードを読み取り、JIT コンパイルとメモリー転送のオーバーヘッドが軽減される可能性があります。一方、モジュールに多くのカーネルが含まれており、そのうちの一部しか使用されない場合は、不要なワークが実行されることになります。未使用のカーネルもロードしてコンパイルする必要があり、メモリー領域を占有します。各ソリューションには賛否両論がありますが、CUDA* (NVCC) はデフォルトでカーネルごとに個別のモジュールを作成するのに対し、DPC++ は複数のカーネルを 1 つの大きなモジュールに結合する傾向があることに注意してください。

これは、数十または数百の多数のテンプレートをインスタンス化する必要のあるテンプレート・カーネルを含む共有ライブラリーを構築する際に問題となる可能性があります。それぞれ 5 つのタイプをサポートし、結果として 125 の組み合わせになる 3 つのテンプレート `typename` パラメーターを持つ関数テンプレートを想像してください。これらすべてのカーネルのデバイスコードが 1 つのモジュールにバンドルされている場合、一般的なユースケースで 1 つの組み合わせしか必要ない場合であっても、これらすべてをロードする必要があります。`nsys` でこのようなプログラムのパフォーマンスを解析すると、CUDA* API セクションの `cuModuleLoadDataEx` 呼び出しで、かなりの時間が費やされていることが示されることがあります。これがアプリケーションのパフォーマンスを制限する場合は、カーネルを別のモジュールに分割する価値があるかもしれません。

DPC++ でのモジュール分割は、`-fsycl-device-code-split` フラグを使用して変更できます。デフォルト値 (`auto`) では、コンパイラーは組み込みのヒューリスティックにより最適な分割を見つけます。結果は、ソース コードの構造と翻訳単位への分割方法によって異なり、意図したユース ケース (上記の共有ライブラリーの例など) にとって常に最適な選択であるとは限りません。このような状況では、`-fsycl-device-code-split=per_kernel` を設定してコンパイラーにカーネルごとに個別のモジュールを生成させると効果的です。

## CUDA バックエンドの共有 (管理) USM

SYCL* 2020 共有 USM 機能 `sycl::malloc_shared` は、ドライバー API `cuMemAllocManaged` を使用して、CUDA* 管理メモリー割り当てと 1:1 でマッピングします。プログラマーは、現在 DPC++ ランタイムには共有 USM の自動管理のため高度なヒューリスティックがないことに注意する必要があります。そのため、共有 USM のデフォルトのパフォーマンスは、`cudaMallocManaged` を使用した同等の CUDA* ランタイム API ほど良くない可能性があります。ただし、ユーザーが手動で `sycl::prefetch` および `sycl::queue::mem_advise` / `sycl::handler::mem_advise` API を使用することは可能です。これらを慎重に使用することで、DPC++ の共有 (管理) USM のパフォーマンスを最適化できます。`sycl::prefetch` および `sycl::queue::mem_advise` (または `sycl::handler::mem_advise`) は、それぞれ `cuMemPrefetchAsync` および `cuMemAdvise` に 1:1 でマップされます。ユーザーは、ハードウェアのプリフェッチとアドバイスを行う際に予想される動作とベスト・プラクティスについて、最新の NVIDIA* ドキュメントを参照する必要があります。次のセクションの目的は、`cudaMemAdvise*` アドバイスと DPC++ の CUDA* バックエンドにおける同等のマッピングについて説明することです。

### mem_advise の CUDA 相当へのマッピング

`sycl::queue::mem_advice` および `sycl::handler::mem_advice` のアドバイス・パラメーターの値と、CUDA* 相当へのマッピングは、次の表に定義されています。

| SYCL* 共有 USM アドバイス | CUDA* 管理メモリーアドバイス | アドバイスの対象 |
|---|---|---|
| `UR_USM_ADVICE_FLAG_SET_READ_MOSTLY` | `cudaMemAdviseSetReadMostly` | キュー/ハンドラーに関連付けられたデバイス |
| `UR_USM_ADVICE_FLAG_CLEAR_READ_MOSTLY` | `cudaMemAdviceUnsetReadMostly` | キュー/ハンドラーに関連付けられたデバイス |
| `UR_USM_ADVICE_FLAG_SET_PREFERRED_LOCATION` | `cudaMemAdviseSetPreferredLocation` | キュー/ハンドラーに関連付けられたデバイス |
| `UR_USM_ADVICE_FLAG_CLEAR_PREFERRED_LOCATION` | `cudaMemAdviseUnsetPreferredLocation` | キュー/ハンドラーに関連付けられたデバイス |
| `UR_USM_ADVICE_FLAG_SET_ACCESSED_BY_DEVICE` | `cudaMemAdviseSetAccessedBy` | キュー/ハンドラーに関連付けられたデバイス |
| `UR_USM_ADVICE_FLAG_CLEAR_ACCESSED_BY_DEVICE` | `cudaMemAdviseUnsetAccessedBy` | キュー/ハンドラーに関連付けられたデバイス |
| `UR_USM_ADVICE_FLAG_SET_PREFERRED_LOCATION_HOST` | `cudaMemAdviseSetPreferredLocation` | ホスト |
| `UR_USM_ADVICE_FLAG_CLEAR_PREFERRED_LOCATION_HOST` | `cudaMemAdviseUnsetPreferredLocation` | ホスト |
| `UR_USM_ADVICE_FLAG_SET_ACCESSED_BY_HOST` | `cudaMemAdviseSetAccessedBy` | ホスト |
| `UR_USM_ADVICE_FLAG_CLEAR_ACCESSED_BY_HOST` | `cudaMemAdviseUnsetAccessedBy` | ホスト |

## 一般的な最適化

ここでは、DPC++ を使用する際の一般的なパフォーマンスの問題や落とし穴、そしてその対処方法について説明します。

### インデックスの入れ替え

SYCL* 仕様の 4.9.1 節では、次のことが規定されています。

> 整数から多次元 id やレンジを構成する場合、多次元空間の線形化において右端の要素が最も速く変化するように要素を記述します。

そのため、インテル® DPC++ コンパイラーでは、右端の次元が CUDA* または HIP の x 次元にマップされ、右から 2 つ目の次元が CUDA* や HIP の y 次元にマップされます。以下に例を示します。

```
cgh.parallel_for(sycl::nd_range{sycl::range(WG_X * WI_X),
sycl::range(WI_X)}, ...)

cgh.parallel_for(sycl::nd_range<2>{sycl::range<2>(WG_Y * WI_Y, WG_X * WI_X),
sycl::range<2>(WI_Y, WI_X)}, ...)

cgh.parallel_for(sycl::nd_range<3>{sycl::range<3>(WG_Z * WI_Z, WG_Y * WI_Y,
WG_X * WI_X), sycl::range<3>(WI_Z, WI_Y, WI_X)}, ...)
```

`WG_X` と `WI_X` は、x 次元の **work-group 数**と **work-group ごとの work-item 数** (CUDA* では、名前付きの**グリッドサイズ**と**ブロックあたりのスレッド数**) であり、`_Y` と `_Z` は y 次元と z 次元のものになります。

次の場合、2 次元または 3 次元のカーネルの `parallel_for` 実行では特に重要であることに注意してください。

- ローカルまたはグローバルメモリー内の (1-d) 配列には、手動で線形化されたアクセスがあります。結合されていないグローバル・メモリー・アクセスまたはローカルメモリー内のバンク競合によるパフォーマンスの問題を回避するため、これを考慮する必要があります。線形化の詳細については、SYCL* 仕様の 3.11 節の多次元オブジェクトと線形化を参照してください。
- 次のエラー (または同等のエラー) が発生します。

  ```
  Number of work-groups exceed limit for dimension 1 : 379957 > 65535
  ```

  これは、CUDA* など一部のプラットフォームでは、x 次元が y および z 次元よりも多くの work-group をサポートするためです。

  ```
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  ```

このトピックの詳細については、こちら (英語) を参照してください。

### 特定の GPU アーキテクチャーの設定

NVIDIA* GPU 向けにビルドする場合、GPU のアーキテクチャーは省略できます。この場合、デフォルトのアーキテクチャーは `sm_50` になります。新しい世代の GPU を使用する場合、適切なアーキテクチャーを指定することでコンパイラーがより高性能な新しい機能を使用できる可能性があります。特定の GPU アーキテクチャーを指定する方法は、「導入ガイド」を参照してください。

### インライン展開

DPC++ は、さまざまなデバイスでパフォーマンスとコンパイル時間のバランスを考慮して、自動的に関数をインライン展開するかどうかを選択します。しかし、プログラマーは、特定の関数に `always_inline` 属性を追加するなどして、インライン展開を強制することもできます。

```
__attribute__((always_inline)) void function(...) {
  ...
}

...

q.submit([&](sycl::handler &cgh) {
    cgh.parallel_for(..., [=](...) {
      function(...);
    });
}
```

関数を手動でインライン展開する場合は、十分注意してください。関数を手動でインライン展開すると、一部の NVIDIA* デバイスではパフォーマンスが向上する可能性がありますが、他の NVIDIA* デバイスではパフォーマンスが低下する可能性があることに注意してください。今後のリリースでは、コンパイラーの最適化ヒューリスティックを改善していく予定です。

### 高速数学ビルトイン

SYCL* 数学ビルトインは、同等の OpenCL* 1.2 数学ビルトインの精度要件と一致するように定義されていますが、一部のアプリケーションでは必要以上に精度が高くなり、パフォーマンスが低下する可能性があります。

これに対処するため、SYCL* 仕様では、数学関数のサブセットのネイティブバージョン (4.17.5 節の「数学関数」に完全なリストがあります) が提供されています。これには、精度とパフォーマンスのトレードオフがあります。これらは、ネイティブ名前空間で定義されています。例えば、`sycl::cos()` のネイティブバージョンは、`sycl::native::cos()` です。

一般に、精度が問題にならない場合、ネイティブバリアントを使用すると大幅に改善できる可能性がありますが、すべてのバックエンドがすべてのビルトインに対し緩和された精度を使用するわけではないことに注意してください。

NVIDIA* アーキテクチャーをターゲットにする場合、次の `sycl::native::` 関数は、対応する PTX 命令の `.approx` バリアントによって実装されます。

- `sycl::native::divide`
- `sycl::native::sqrt`
- `sycl::native::sin`
- `sycl::native::cos`

- `sycl::native::log2`
- `sycl::ext::oneapi::experimental::native::exp2`
- `sycl::ext::oneapi::experimental::native::tanh`

上記 `sycl::native::` 数学関数と PTX 命令は *1:1* でマッピングされます。例えば、1 つの `sycl::native::exp2` 呼び出しに対し、コンパイラーは 1 つの `ex2.approx` 命令を生成します。別のケースでは、ネイティブ数学関数は、複数の `.approx` PTX 命令によって実装されます。例えば、`sycl::native::tan()` は、`sin.approx`、`cos.approx`、および `divide.approx` で実装されます。ネイティブ数学関数は、精度と引き換えに `sycl::` の対応する関数よりも高速になる可能性があります。`.approx` PTX 命令の精度の詳細については、PTX ISA ドキュメント (英語) を参照してください。

**注意:** `-ffast-math` コンパイルオプションは、標準の `sycl::` 数学関数を、対応する `sycl::native::` 関数に入れ替えます (利用可能であれば)。指定された数学関数のネイティブバージョンが存在しない場合、`-ffast-math` フラグは影響しません。

`icpx` コンパイラーでは `-ffast-math` がデフォルトです。`icpx` で `-ffast-math` を無効にするには、`-fno-fast-math` を使用します。

**ループアンロール**

コンパイラーは一部のループアンロールを自動的に行いますが、次のように `unrolling` プラグマを使用して、デバイスコード内の計算集約型ループのアンロールを手動でコンパイラーに指示することが有益な場合もあります。

```
#pragma unroll <unroll factor>
for( ... ) {
 ...
}
```

関数を手動でインライン展開する場合は、十分注意してください。関数を手動でインライン展開すると、一部の NVIDIA* デバイスではパフォーマンスが向上する可能性がありますが、他の NVIDIA* デバイスではパフォーマンスが低下する可能性があることに注意してください。今後のリリースでは、コンパイラーの最適化ヒューリスティックを改善していく予定です。

**インデックスのダウンキャスト**

SYCL* では、`nd_range`、`range`、およびその他のインデックス・タイプは、値タイプとして `size_t` を使用します。これは、`threadIdx.{x|y|z}` が `int` である CUDA* や HIP などのバックエンド API とは対照的です。カーネル内のレジスタースペースを節約するには、インデックス計算関数から返される `size_t` から、バックエンド API がインデックス作成に使用するタイプ (通常は `int`) にインデックス・タイプをダウンキャストすると効果的です。

```
auto bigIdx = item.get_id(0); // size_t
int intIdx = static_cast<int>(item.get_id(0));
```

これは、レジスター・プレッシャーが高いカーネルでは特に有益です。

`get_linear_id` や `get_global_linear_id` などのメンバー関数は、SYCL* 実装では、異なる `size_t` インデックスが次のような方法で結合されます。

```
size_t item<2>::get_linear_id() {
  return get_id(0) * get_range(1) + get_id(1);
}
```

SYCL* は、32 ビット・タイプの使用時に発生するオーバーフローを回避するため、このような計算に `size_t` を使用するように指定されています。演算は `get_range` および `get_id` によって返される `size_t` タイプで行われるため、このメンバー関数の結果を `int` または他のタイプにキャストしても、関数の実装内で `size_t` の使用が排除されるわけではありません。したがって、プログラマーがこれらの計算に 32 ビット・タイプ (またはそれ以下) のみを使用する場合は、線形インデックス計算を抽象化する呼び出しを省略する必要があります。次のような `unsafe_get_linear_id` の実装は、インデックス計算から 64 ビット・サイズのタイプをすべて削除します。

```
inline int unsafe_get_linear_id(item<2> it) {
  return static_cast<int>(it.get_id(0)) * static_cast<int>(it.get_range(1))
              + static_cast<int>(it.get_id(1));
}
```

これにより、インデックス計算用の SYCL* コードのデバイスコードのフットプリントがネイティブ CUDA* または HIP コードと同じになります。

上記の例では、プログラマーが整数オーバーフローの発生を防ぐ責任を負うため、安全ではないと考えられます。

**アクセサーの使用法を最適化**

SYCL* アクセサーは、デバイスメモリーへのインデックス付けを行う高レベルのインターフェイスをユーザーに提供します。アクセサーは `accessOffset` 引数を使用して構築できます。これにより、アクセサーのベース・インデックスを、アクセサーが参照するメモリーの実際のベース・インデックスからオフセットすることができます。この機能は、クリーンで移植性の高いコードを記述する際に役立ちますが、すべての `operator[]` 呼び出しに対して `accessOffset` パラメーターを使用して `operator[]` を計算する必要があるため、`acc[idx]` の計算が最適ではなくなる可能性があります。

```
T &operator[](size_t idx) {
  return data[idx + accessOffset];
}
```

これにより、カーネルのセットアップ時間とレジスターの使用量が増加し、パフォーマンスが低下する可能性があります。ユーザーがアクセサーを使用しながらこのオーバーヘッドを排除したい場合は、`get_multi_ptr()` メソッドを使用してアクセサーのデータにインデックスを付けることを推奨します。`get_multi_ptr()` は常に割り当てのベースポインターを返すため、`accessOffset` によってオフセットされることはありません。

```
// idx と accessOffset を内部的に結合しません
auto refVal = acc.get_multi_ptr()[idx];
```

このパターンを使用すると、レジスターの使用量とカーネルのセットアップ時間が削減されます。

これはアクセサークラスのみの問題であり、USM ポインターへのインデックス作成時には発生しないことに注意してください。

### ローカルアクセサー

SYCL* 2020 は、SYCL* プログラムでローカルメモリーを使用するための `local_accessor` クラスを提供しています。これは便利であり、デバイスメモリーのアクセサーモデルを反映しています。ただし、一部のアプリケーションでは、特定の最適化が抑制され、実行時のパフォーマンスが低下する可能性があります。次のコード例を検討してください。

```
template <int lmem_size>
void MatrixMul(float *C, float *A, float *B, int wA, int wB,
               sycl::nd_item<3> item_ct1,
               sycl::local_accessor<float, 2> As,
               sycl::local_accessor<float, 2> Bs) {
// continues
```

ローカルアクセサーのサイズは、`queue::submit` スコープ内のワーク・グループ・サイズと一致するように静的に設定されます。ただし、コンパイラーはホストとデバイスにまたがってこれを認識できないため、アクセサーサイズから想定することができません。そのため、多くの最適化の可能性を逃す可能性があります。

インテルは、ユーザーがカーネル内で別の方法でローカルメモリーを指定できるように SYCL* 仕様の拡張機能を公開しました。

この拡張機能を使用すると、次のように記述できます。

```
template <int lmem_size>
void MatrixMul(float *C, float *A, float *B, int wA, int wB,
               sycl::nd_item<3> item_ct1 {
  using namespace sycl::ext::oneapi;
  auto& As =
*group_local_memory_for_overwrite<float[lmem_size][lmem_size]>(item_ct1.get_gr
oup());
  auto& Bs =
*group_local_memory_for_overwrite<float[lmem_size][lmem_size]>(item_ct1.get_gr
oup());
```

この場合、ベースとなる配列が保持するタイプとして `float[size][size]` を使用しますが、静的にサイズ設定された任意の配列タイプにすることができます。コンパイル時に割り当てのサイズが判明していると、コンパイラーはコードを推論でき、例えば、デバイスのグローバルメモリーからローカルメモリーにタイル化されたデータをロードするときに、適切なコード生成が可能になります。これがいつ起こるかを判断するのは難しいため、これまでと同様に、生成されたコードを調査し、プログラムでパフォーマンス・テストを実行することが、ローカルアクセサーよりも使用するタイミングを判断する最善の方法です。

### エイリアス解析

エイリアス解析では、2 つのメモリー参照が互いにエイリアスでないことが証明できます。これにより最適化が有効になることがあります。デフォルトでは、コンパイラーはエイリアス解析によって証明されない限り、メモリー参照はエイリアスであると想定する必要があります。ただし、デバイスコード内のメモリー参照がエイリアスではないことをコンパイラーに明示的に通知することもできます。これは、バッファー/アクセサーと USM モデルのそれぞれのキーワードを使用することで実現できます。

前者は、oneapi 拡張の no_alias プロパティーをアクセサーに追加することができます。

```
q.submit([&](sycl::handler &cgh) {
  sycl::accessor acc{...,
sycl::ext::oneapi::accessor_property_list{sycl::ext::oneapi::no_alias}};
  ...
});
```

後者の場合、__restrict__ 修飾子をポインターに追加できます。

__restrict__ は C++ では非標準であり、SYCL* 実装全体で一貫性がない可能性があることに注意してください。dpc++ では、restrict 修飾されたデバイス関数 (SYCL* カーネルから呼び出される関数) パラメーターのみが考慮されます。

例:

```
void function(int *__restrict__ ptr) {
    ...
}

...
int *ptr = sycl::malloc_device<int>(..., q);
...
q.submit([&](sycl::handler &cgh) {
      cgh.parallel_for(..., [=](...) {
        function(ptr);
      });
});
```

より強制的なアプローチは、[[intel::kernel_args_restrict]] 属性をカーネルに追加することです。これは、各 USM ポインター間、またはそのモデルがカーネル内で使用される場合はバッファーアクセサー間のすべてのエイリアス依存関係を無視するようにコンパイラーに指示します。

例 (バッファー/アクセサーモデル):

```
q.submit([&](handler& cgh) {
      accessor in_accessor(in_buf, cgh, read_only);
      accessor out_accessor(out_buf, cgh, write_only);
        cgh.single_task<NoAliases>([=]() [[intel::kernel_args_restrict]] {
          for (int i = 0; i < N; i++)
              out_accessor[i] = in_accessor[i];
        });
});
```

CUDA* プラットフォームでは、sycl_ext_oneapi_cuda_tex_cache_read 拡張 (英語) の ldg テンプレート関数を使用することも、メモリーアクセスのパフォーマンス向上に役立ちます。

**テクスチャー・キャッシュの使用**

CUDA* プラットフォームでは、少なくともカーネルの存続期間中は一定であるデータをテクスチャー・キャッシュにキャッシュできます。

これは、`sycl::ext::oneapi::experimental::cuda::ldg` 関数を使用して実現できます。この関数は、デバイスメモリーへのポインターを受け取り、L1/tex キャッシュからロードして、アドレスに格納された値を返します。以下に例を示します。

```
float some_value = ldg(&some_data_in_device_memory[some_index]);
```

**警告:** この関数でロードされたデータがカーネル内に書き込まれることをコンパイラーが検出した場合でも、プログラムはコンパイルできますが、テクスチャー・キャッシュは使用されないことに注意してください。

テクスチャー・キャッシュを使用することでパフォーマンス向上に影響する要因は数多くあります。そのため、最大の高速化を達成するのは困難な場合があります。実際、多くのユースケースではメリットがほとんどないか、全くありません。ただし、パフォーマンスが低下する可能性は低く、低下した場合でも規模は小さく、`ldg` は最小限のコード変更で使用できるため、カーネルのパフォーマンスを素早く向上する素晴らしい方法となるかもしれません。

`ldg` は、HIP AMD を含むほかのすべてのプラットフォームでも移植可能であることに注意してください。しかし、CUDA* は現在 `ldg` により特殊なキャッシングが可能な唯一のプラットフォームです。HIP AMD バックエンドは、`ldg` を使用するかどうかにかかわらず、常にすべてのレジスターデータを L1 キャッシュと L2 キャッシュにロードします。

テクスチャー・キャッシュの詳細については、こちらのブログ (英語) を参照してください。`ldg` 関数の詳細は、対応する拡張機能のドキュメント (英語) をご覧ください。

### AMDGPU 安全でないアトミック

`malloc_device` 割り当てで実行される AMD* GPU のアトミック操作は、パフォーマンスを向上させるため、安全でないアトミックを使用することをお勧めします。これは、`int` タイプのアトミックの場合は `-mllvm --amdgpu-oclc-unsafe-int-atomics=true`、`fp` タイプのアトミックの場合は `-mllvm --amdgpu-oclc-unsafe-fp-atomics=true` オプションを追加することで実行できます。ただし、この場合の安全でないアトミックのサポートは PCI Express のバージョンに依存し、新しい世代のみがこの新しい高速命令をサポートするため、この方法は `malloc_shared` 割り当てでは機能しない可能性があります。

したがって、デフォルトでは、PCIe* がサポートする安全な CAS (比較とスワップ) アトミックが使用されます。CAS アトミックは、安全でない同等のアトミックよりもはるかに低速であるため、最高のパフォーマンスを得るには、安全でないアトミックフラグを指定した `malloc_device` 割り当てを使用することをお勧めします。両方のフラグはグローバルオプションであり、ユーザーコードに同じ TU 内に `malloc_shared` アトミックと `malloc_device` アトミックの両方が含まれている場合は使用してはなりません。

# サポート

## 機能

### コア機能

| 機能 | サポート |
|---|---|
| コンテキスト内の複数デバイス | はい |
| サブグループ (sub-group) | はい |
| グループ関数/アルゴリズム | はい |
| 整数関数 | はい |
| 数学関数 (スカラー) | はい |
| 数学関数 (ベクトル) | はい |
| 数学関数 (marray) | はい |
| 共通関数 | はい |
| ジオメトリー関数 | はい |
| リレーショナル関数 | はい |
| atomic ref | はい |
| オペレーティング・システム | Linux* |
| バッファーの再解釈 | はい |
| stream | はい |
| デバイスイベント | はい |
| グループの非同期コピー | はい |
| プラットフォームの get info | はい |
| カーネルの get info | はい |
| sycl::nan と sycl::isnan | はい |
| デバイスセレクター | はい |
| 階層的並列化 | はい |
| ホストタスク | はい |
| インオーダー・キュー | はい |
| リダクション | はい |
| キューのショートカット | はい |
| vec | はい |
| marray | はい |
| errc | はい |
| 匿名カーネルラムダ | はい |
| 機能を評価するマクロ | はい |

| 機能 | サポート |
| --- | --- |
| sycl::span | はい |
| sycl::dynamic_extent | いいえ[1] |
| sycl::bit_cast | はい |
| aspect_selector | いいえ |
| カーネルバンドル | はい |
| 特殊化定数 | はい |

**非コア機能**

| 機能 | サポート |
| --- | --- |
| image | いいえ (ext_oneapi_bindless_images の場合、はい) |
| fp16 データタイプ | はい |
| fp64 データタイプ | はい |
| prefetch | はい |
| USM | ホスト、デバイス、共有 |
| USM アトミックホスト割り当て | いいえ |
| USM アトミック共有割り当て | はい |
| USM システムに割り当て | はい |
| SYCL_EXTERNAL | はい |
| アトミックメモリーの順序付け | relaxed、acquire、release、acq_rel、seq_cst |
| アトミック・フェンス・メモリーの順序付け | はい |
| アトミック・メモリー・スコープ | sub_group、work_group、device、system |
| アトミック・フェンス・メモリーのスコープ | はい |
| 64 ビット・アトミック | はい |
| バイナリー形式 | NVPTX と SASS |
| デバイスのパーティション化 | いいえ |
| ホストデバッグ可能デバイス | いいえ |
| オンラインコンパイル | いいえ |
| オンラインリンカー | はい |
| キューのプロファイル | はい |
| mem_advise | read_mostly、優先場所、アクセス元 |
| バックエンド仕様 | WIP[2] |
| アプリケーション・バックエンドの相互運用 | 部分的[2] |
| カーネル・バックエンドの相互運用 | いいえ |
| ホストタスク (ハンドルと相互運用) | はい |
| reqd_work_group_size | はい |

| 機能 | サポート |
| --- | --- |
| キャッシュビルド結果 | いいえ |
| ビルドログ | いいえ |
| ビルトインカーネル関数 | なし |

**拡張機能**

このリリースに対応するすべての DPC++ 拡張機能の詳細については、拡張機能のドキュメント (英語) を参照してください。

| 機能 | サポート |
| --- | --- |
| uniform | いいえ |
| USM アドレス空間 (デバイス、ホスト) | 部分的 |
| 固定ホストメモリーの使用 | はい |
| サブグループ・マスク (+ グループ投票) | はい |
| 静的ローカルメモリー使用量照会 | いいえ |
| sRGB イメージ | いいえ |
| デフォルト・プラットフォーム・コンテキスト | 一部分 |
| メモリーチャネル | いいえ |
| 最大ワークグループ照会 | 一部分 |
| 結合行列 | はい |
| すべて制限 (restrict all) | いいえ |
| プロパティー・リスト (property list) | いいえ |
| カーネル・プロパティー (kernel properties) | いいえ |
| SIMD 呼び出し | いいえ |
| 低レベルデバイス情報 | いいえ |
| カーネルキャッシュ設定 | いいえ |
| FPGA lsu | いいえ |
| FPGA reg | いいえ |
| データ・フロー・パイプ | いいえ |
| キューに投入されたバリア | いいえ |
| フィルターセレクター | はい |
| グループソート | はい |
| フリー関数の照会 | はい |
| 明示的な SIMD | いいえ |
| discard_queue_events | はい |
| device_if | いいえ |
| device_global | はい |

| 機能 | サポート |
| --- | --- |
| C と C++ 標準ライブラリーのサポート | はい (clang++ 使用時) |
| カーネルでの assert | はい |
| buffer_location | いいえ |
| accessor_property_list (+ no_offset、no_alias) | はい |
| グループ・ローカル・メモリー | はい |
| printf | はい |
| ext_oneapi_bfloat16 | はい |
| 拡張デバイス情報 | いいえ |
| sycl_ext_oneapi_cuda_tex_cache_read | はい |
| sycl_ext_oneapi_native_math | はい |
| sycl_ext_oneapi_bfloat16_math_functions | はい |
| sycl_ext_oneapi_cuda_async_barrier | はい |
| sycl_ext_oneapi_bindless_images | はい |
| sycl_ext_oneapi_graph | 実験的 |
| sycl_ext_oneapi_non_uniform_groups | はい (tangle_group を除く) |
| sycl_ext_oneapi_peer_access | はい |
| sycl_ext_codeplay_enqueue_native_command | はい |

[1] numeric_limits<size_t>::max() の使用

[2] (1、2) https://github.com/KhronosGroup/SYCL-Docs/pull/197 (英語) を参照

## 更新履歴

### 2025.0.0

NVIDIA* プラグインが Windows* で利用できるようになりました。

#### 改良点

- sm90a アーキテクチャーを追加しました [f204869]。
- PTX バージョンを提供するため `__PTX_VERSION__` マクロを追加しました [58f829a] 。
- `sycl_ext_codeplay_enqueue_native_command` 拡張を実装しました [0f48227] 。
- カーネル内の C-CXX 標準ライブラリーのサポートを改善しました [da379ec、9942378] 。
- 不正な NVIDIA* トリプルに対するコンパイル診断を改善しました [4240ef0] 。
- レジスターが不足した際のエラーメッセージを改善しました [9f1cee57] 。
- SYCL-Graph のフィルノード実装を改善しました [統合ランタイム bb589ca] 。
- SYCL-Graph のノード依存関係を強制する空のノードとして `handler::prefetch` と `handler::mem_advise` を実装しました [統合ランタイム 3c12bbc] 。
- SYCL-Graph エッジの実装に使用される UR 同期ポイントからオーバーヘッドの一部を削除しました [unified-runtime 3c12bbc] 。
- すべての NVIDIA* デバイスに 1 つのプラットフォームを使用します [統合ランタイム f05c1c8] 。
- Bindless Images のキューブマップと画像配列をサポートしました [83bbea926ae7、99635a0d214b] 。
- テクスチャー・フェッチ機能を追加しました [d13fdbe4ee02] 。
- Bindless Images のデバイス間コピーのサポートを追加しました [統合ランタイム f4898299] 。
- Bindless Images の DirectX* 12 相互運用性 [bd97f283c9f9、統合ランタイム 487f4f8a] 。

#### バグフィックス

- `nextafter(-0.0,+0.0)` を修正しました [d6780ae7]
- 非均一なグループシャッフルの問題を修正しました [a0c3b325]
- `queue.fill()` 使用時のパフォーマンスの問題を修正しました [0ccb0b7]
- `nullptr` の `multi_ptr` 関係演算子を修正しました [4f91bbb]
- CUDA* ストリーム作成時の競合状態を修正しました [統合ランタイム cabf128]

### 2024.2.0

#### 改良点

- 連続的に一貫性のあるメモリー順序をサポートしました (`sm_70+`) [c1e2957]。

#### バグフィックス

- フェンス実装を修正して SYCL* 2020 セマンティクスに一致させました [95e183e6]。
- ローカル・ワーク・サイズの推測を修正および改善しました [unified-runtime 43f0963]。

### 2024.1.0

#### 改良点

- `sycl_ext_oneapi_graph` をサポートしました [367b662a]。
- `sycl_ext_oneapi_device_architecture` をサポートしました [1ad69e59]。
- `ext_oneapi_queue_priority` をサポートしました [0c33fea5]。
- 正規化されたチャネルタイプのサポートを追加しました [fd5014ad]。
- エラーコードの戻り値の関連性を改善しました [67a24f7b、b7a43a42]。

#### バグフィックス

- 報告された最大ローカル・メモリー・サイズを修正しました [d2719b5]。
- `-fgpu-rdc` オプションを修正しました [f7595ac]。
- `rintf`、`nearbyint` の欠落を修正しました [3c327c73、0ef26d3e]。
- イベント・プロファイルの競合状態を修正しました [e8ffd021]。

#### 非推奨

- コンテキストの相互運用性が非推奨となり、代わりにプライマリー・コンテキストを使用する必要があります [e213fe2f]。

### 2024.0.2

- 変更ありません。

### 2024.0.1

- `sycl_ext_oneapi_bindless_images` (英語) 拡張機能をサポートしました。

### 2024.0

#### 改良点

SYCL* コンパイラー

- `sycl::range` を使用する `parallel_for` インターフェイスの `work_group` サイズの選択が改善されました。

SYCL* ライブラリー

- sycl_ext_oneapi_graph 拡張がサポートされました。
- sycl_ext_oneapi_non_uniform_groups 拡張がサポートされました。
- sycl_ext_oneapi_peer_access 拡張がサポートされました。
- max_registers_per_work_group デバイスクエリーが導入されました。
- `SYCL_PROGRAM_COMPILE_OPTIONS` を使用するメカニズムが追加され、`maxrregcount` ptxas コンパイラー・オプションを次のように CUDA* バックエンドに渡すことができるようになりました。

```
SYCL_PROGRAM_COMPILE_OPTIONS="--maxrregcount=<value>"
```

これは、JIT コンパイルされたプログラムに対してのみ機能することに注意してください。

**バグフィックス**

- 倍精度浮動小数点グループ・アルゴリズムが修正され、`icpx` コンパイラーでコンパイルしてもハングアップしなくなりました。

  `broadcast`、`joint_exclusive_scan`、`joint_inclusive_scan`、`exclusive_scan_over_group`、`inclusive_scan_over_group`

- 引数として `sycl::memory_scope::device` を渡したときに、誤った `memory_scope` を使用していた `atomic_fence` のバグを修正しました。

**2023.2.0**

**改良点**

SYCL* コンパイラー

- NVPTX バックエンドのインライン展開のしきい値乗数を増やすことで clang++/cuda のパフォーマンスが向上しました [22d98280]。
- SYCL* の `__CUDA_ARCH__` の代わりに `__SYCL_CUDA_ARCH__` を定義しました [8f5000c3]。
- SYCL* ライブラリー

SYCL* ライブラリー

- `__ldg*` clang ビルトインを CUDA* 専用の拡張機能として SYCL* に公開する `sycl_ext_oneapi_cuda_tex_cache_read` を導入しました – 読み取り専用テクスチャー・キャッシュ (英語) [5360825e]。
- `cl_khr_subgroups` 拡張機能をサポートするサブグループとしてレポートするようになりました [8e6c092b]。
- `atomic_fence` デバイスクエリーは、NVIDIA によってドロップされる可能性があるエラー [82ac98f8] で失敗せずに、必要最小限の機能を返すようになりました [1e88df54]。
- 理論上のピークメモリー帯域幅のクエリーをサポートしました – インテルのデバイス情報拡張機能 (英語) [8ce0a6d5]
- デバイス ID と UUID のサポートを追加しました – インテルのデバイス情報拡張機能 (英語) [8213074d]
- ホストデバイス `memcpy2D` をサポートしました [d0b25d4a]。
- CUDA* バックエンドで `sycl_ext_oneapi_memcpy2d` をサポートしました - oneAPI memcpy2d (英語) [9008a5d2]

**バグフィックス**

- 無効な work-group サイズに関するエラーを `PI_ERROR_INVALID_WORK_GROUP_SIZE` に置き換えるようになりました [2357af0a]。
- `sycl::ctz` 関数からの間違った結果に対応しました [5a9f601e]。
- イベントが意図したとおりに待機しない原因となる問題に対処しました [1b225447]。

### 2023.1.0

#### 改良点

SYCL* コンパイラー

- FTZ、`prec-sqrt` が `no-ftz`、`no-prec-sqrt` をオーバーライドできるようになりました [8096a6fb]。
- `-fsycl-targets` の引数として NVIDIA* アーキテクチャー (`nvidia_gpu_sm_80` など) を指定できるようになりました [e5de913f]。

SYCL* ライブラリー

- 新しい「unified」インターフェイスによる行列拡張を実装しました [166bbc36]。
- CUDA* バックエンドのゼロ・レンジ・カーネルをサポートしました [a3958865]。
- 不足しているマクロを `interop-backend-traits.cpp` に追加しました [a578c8141]。
- CUDA* バックエンドで各種メタデータを許可するようになりました [25d05f3d]。
- `tf32` デバイスコードのチェックコメントを更新しました [21176576]。

#### バグフィックス

- `ext_oneapi_cuda make_device` が `sycl::device` を複製しなくなりました [75302c53a]。
- ガードが正しく構築されない問題を修正しました [ce7c594f]。

#### ドキュメント

- `ptxas` オプションを渡す方法がドキュメント化されました [f48f96eb3f]。
- `cuda-arch` 固有の機能を有効にする CUDA* GPU アーキテクチャーの説明が追加されました [4e5d276f]。

### 2023.0.0

oneAPI for NVIDIA* GPU の最初のリリースです。

このリリースは、intel/llvm repository at commit 0f579ba (英語) から作成されました。

#### 新機能

- CUDA* バックエンドのサポート

#### SYCL* コンパイラー

- `sycl::half` タイプのサポート
- ストレージタイプで動作する `bf16` ビルトインのサポート
- リレーショナル、ジオメトリー、共通、および数学カテゴリーからの SYCL* ビルトインのサポート
- sub_group 拡張のサポート
- グループ・アルゴリズムのサポート
- `group_ballot` 組込み関数のサポート

- スコープとメモリー順序によるアトミックのサポート
- 同時実行を改善する各キューでの複数ストリームのサポート
- `sycl::queue::mem_advise` のサポート
- CUDA* libclc での `--ffast-math` のサポート
- デバイスでの `assert` をサポート
- CUDA* libclc での float/double 変換、比較交換のアトミック操作をサポート
- CXX 標準ライブラリー関数に対応
- デフォルト `ctor` の `sycl::event` のネイティブイベントは COMPLETE 状態

**SYCL* ライブラリー**

- `fma`、`fmin`、`fmax` および `fmax` に `bf16` ビルトインを追加
- `sycl::aspect::fp16` をサポート
- `tanh` (float/half) と `exp2` (half) のネイティブ定義を追加
- `sycl::get_native(sycl::buffer)` をサポート
- `mem_advise` リセットと同時メモリーチェック管理を実装
- `bfloat16` のサポートを含む、`joint_matrix` での要素ごとの操作をサポート
- 統合共有メモリー (USM) をサポート

## トラブルシューティング

この節では、トラブルシューティングのヒントと一般的な問題の解決方法について説明します。ここで説明する方法で問題が解決しない場合は、Codeplay のコミュニティー・サポート・ウェブサイト (英語) からサポートリクエストをお送りください。完全なサポートは保証できませんが、できる限り支援させていただきます。サポートリクエストを送信する前に、ソフトウェアが最新の安定したバージョンであることを確認してください。

問題、パフォーマンス、機能要望は、oneAPI DPC++ コンパイラーのオープンソース・リポジトリー (英語) からも報告できます。

**sycl-ls の出力にデバイスが表示されない**

`sycl-ls` がシステム上の期待されるデバイスを報告しない場合:

1. システムに互換性のあるバージョンの CUDA* または ROCm* ツールキット (それぞれ CUDA* と HIP プラグイン向け)、および互換性のあるドライバーがインストールされていることを確認してください。
2. `nvidia-smi` または `rocm-smi` がデバイスを正しく認識できることを確認します。
3. プラグインが正しくロードされていることを確認します。これは、環境変数 `SYCL_PI_TRACE` に `1` を設定して、`sycl-ls` を再度実行することで分かります。

   例:

   ```
   $ SYCL_PI_TRACE=1 sycl-ls
   ```

次のような出力が得られるはずです。

```
SYCL_PI_TRACE[basic]: Plugin found and successfully loaded: libpi_opencl.so
[ PluginVersion: 11.15.1 ]
SYCL_PI_TRACE[basic]: Plugin found and successfully loaded:
libpi_level_zero.so [ PluginVersion: 11.15.1 ]
SYCL_PI_TRACE[basic]: Plugin found and successfully loaded: libpi_cuda.so
[ PluginVersion: 11.15.1 ]
[cuda:gpu][cuda:0] NVIDIA CUDA BACKEND, NVIDIA A100-PCIE-40GB 8.0 [CUDA
12.5]
```

インストールしたプラグインが `sycl-ls` の出力に表示されない場合、`SYCL_PI_TRACE` に `-1` を設定して再度実行することで、詳細なエラー情報を取得できます。

```
$ SYCL_PI_TRACE=-1 sycl-ls
```

大量の出力が得られますが、次のようなエラーが表示されているか確認してください。

```
SYCL_PI_TRACE[-1]:
dlopen(/opt/intel/oneapi/compiler/2024.2.0/linux/lib/libpi_hip.so) failed
with <libamdhip64.so.4: cannot open shared object file: No such file or
directory>
SYCL_PI_TRACE[all]: Check if plugin is present. Failed to load plugin:
libpi_hip.so
```

- CUDA* プラグインには、CUDA* SDK で提供される `libcuda.so` と `libcupti.so` が必要です。
- HIP プラグインには、ROCm* の `libamdhip64.so` が必要です。

CUDA* または ROCm* のインストールと、環境が適切に設定されていることを確認してください。また、`LD_LIBRARY_PATH` が上記のライブラリーを検出できる場所を指しているか確認してください。

4. `ONEAPI_DEVICE_SELECTOR` または `SYCL_DEVICE_ALLOWLIST` などのデバイスフィルター環境変数が設定されていないことを確認します (`ONEAPI_DEVICE_SELECTOR` が設定されていると、`sycl-ls` は警告を表示します)。
5. 権限を確認します。POSIX* では、アクセラレーター・デバイスへのアクセスは、通常、適切なグループのメンバーであることを条件としています。例えば、Ubuntu* Linux* の場合、GPU へのアクセスには `video` グループと `render` グループのメンバーである必要がありますが、これは設定によって異なります。

**不正バイナリーエラーの扱い**

**不適切なプラットフォーム**

よくある間違いは、SYCL* プログラムに互換性のあるバイナリーがないプラットフォームを使用して SYCL* プログラムを実行することです。例えば、SYCL* プログラムは SPIR-V* バックエンド用にコンパイルされた後、HIP デバイス上で実行される可能性があります。この場合、`PI_ERROR_INVALID_BINARY` エラーコードがスローされます。この場合、次の点を確認してください。

1. プログラムが適切なプラットフォーム向けにコンパイルされるよう、`-fsycl-targets` にターゲット・プラットフォームが指定されていることを確認します。

2. プログラムが、実行可能ファイルがコンパイルされたプラットフォームと互換性のある SYCL* プラットフォームまたはデバイスセレクターを使用していることを確認します。環境変数 `SYCL_PI_TRACE=1` を指定して実行し、実行時に選択されたデバイスを表示します。

**適切なプラットフォームと不適切なデバイス**

CUDA* または HIP をターゲットにする SYCL* アプリケーションを実行すると、特定の状況でアプリケーションが失敗し、無効なバイナリーであることを示すエラーが報告されることがあります。例えば、CUDA* の場合は `CUDA_ERROR_NO_BINARY_FOR_GPU` がレポートされる場合があります。

これは、選択された SYCL* デバイスに適切でないアーキテクチャーのバイナリーが送信されたことを意味します。この場合、次の点を確認してください。

1. アプリケーションが、利用するハードウェアのアーキテクチャーと一致するようにビルドされていることを確認してください。
    - CUDA* 向けのフラグ:

      ```
      -Xsycl-target-backend=nvptx64-nvidia-cuda --cuda-gpu-arch=<arch>
      ```

    - HIP 向けのフラグ:

      ```
      -Xsycl-target-backend=amdgcn-amd-amdhsa --offload-arch=<arch>
      ```

2. 実行時に適切な SYCL* デバイス (ビルドされたアプリケーションのアーキテクチャーに一致するもの) が選択されていることを確認します。環境変数 `SYCL_PI_TRACE=1` を設定すると、選択されたデバイスに関連するトレース情報を表示できます。以下に例を示します。

   ```
   SYCL_PI_TRACE[basic]: Plugin found and successfully loaded: libpi_opencl.so
   [ PluginVersion: 11.16.1 ]
   SYCL_PI_TRACE[basic]: Plugin found and successfully loaded:
   libpi_level_zero.so [ PluginVersion: 11.16.1 ]
   SYCL_PI_TRACE[basic]: Plugin found and successfully loaded: libpi_cuda.so
   [ PluginVersion: 11.16.1 ]
   SYCL_PI_TRACE[all]: Requested device_type: info::device_type::automatic
   SYCL_PI_TRACE[all]: Requested device_type: info::device_type::automatic
   SYCL_PI_TRACE[all]: Selected device: -> final score = 1500
   SYCL_PI_TRACE[all]:   platform: NVIDIA CUDA BACKEND
   SYCL_PI_TRACE[all]:   device: NVIDIA GeForce GTX 1050 Ti
   ```

3. 誤ったデバイスが選択されている場合、環境変数 `ONEAPI_DEVICE_SELECTOR` を使用して SYCL* デバイスセレクターが選択するデバイスを変更できます。インテル® oneAPI DPC++/C++ コンパイラーのドキュメントにある「環境変数」の節を参照してください。

**外部参照関数「…」を解決できません/外部シンボル「…」が未定義です**

これにはいくつかの原因が考えられます。

1. 現在 DPC++ では `std::complex` はサポートされていません。代わりに `sycl::complex` を使用してください。

2. `icpx` コンパイラーは、デフォルトで `-ffast-math` モードを使用するため、現在 `ldexp` や `logf` などの特定の数学関数の解決に問題が生じることがあります。これは、`-fno-fast-math` フラグを使用して `-ffast-math` を無効にすることで回避できます。

   詳細は、「oneAPI for NVIDIA* GPU のインストール」を参照してください。

**コンパイラーのエラー:「cannot find libdevice (libdevice が見つかりません)」**

CUDA* SDK がデフォルトの位置にインストールされていないと、`clang++` が SDK を検出できず、コンパイル中に次のようなエラーが発生することがあります。

```
clang-17: error: cannot find libdevice for sm_50; provide path to different
CUDA installation via '--cuda-path', or pass '-nocudalib' to build without
linking with libdevice
```

この問題を解決するには、`--cuda-path` オプションで CUDA* SDK のインストール・パスを指定します。

**コンパイラーのエラー:「needs target feature (ターゲットの機能が必要です)」**

DPC++ ランタイムで使用される一部の nvptx ビルトインは、コンパイルに最小限の計算機能を要求します。プログラムが使用するビルトインに対し十分な計算機能をターゲットにしていない場合 (コンパイル引数 `-Xsycl-target-backend --cuda-gpu-arch=sm_xx` を使用)、次のエラーが報告されます。

```
error: '__builtin_name' needs target feature (sm_70|sm_72|..),...
```

このようなエラーを回避するには、十分な計算機能を持つデバイスをターゲットにしてコンパイルしていることを確認してください。サポートされている計算機能を持つデバイスをコンパイラーに指定してもこのようなエラーが発生する場合、`-Xsycl-target-backend` に 32 ビット・トリプル `nvptx-NVIDIA-cuda` を渡していると考えられます。`nvptx-NVIDIA-cuda` トリプルは、ターゲット機能のビルトインをコンパイルできず、DPC++ では公式にサポートされていません。64 ビット・トリプル `nvptx64-NVIDIA-cuda` は、最近の NVIDIA* デバイスをすべてサポートするため、こちらを使用することを推奨します。

**コンパイラーの警告:「CUDA version is newer than the latest supported version (CUDA* のバージョンがサポートされるバージョンよりも新しいです)」**

このリリースでは、使用する CUDA* のバージョンによっては、コンパイラーが次のような警告を出力することがあります。

```
clang++: warning: CUDA version is newer than the latest supported version 11.5
[-Wunknown-cuda-version]
```

通常、この警告は無視してもかまいません。DPC++ は、最新の CUDA* でサポートされる機能を使用しないかもしれませんが、大部分のシナリオでは問題なく動作するはずです。

### カーネル起動時のリソース不足

以下のエラーコードのいずれかが表示される場合、適切なリソースが不足しているため起動できなかった可能性があります。

- `CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES`
- `CUDA_ERROR_INVALID_VALUE`
- `PI_ERROR_INVALID_WORK_GROUP_SIZE`
- `PI_ERROR_INVALID_VALUE`

### 考えられる理由

- デバイスの work-item (CUDA* スレッド) の最大数を超えています。
- デバイスの最大 work-group サイズ (CUDA* ではスレッドブロック) を超えています。
- カーネルのリソース (レジスターや共有メモリー) がデバイスの能力を超えています。

デバイスの能力をチェックしてこれらの可能性を検証し、制限要因を考慮してカーネル起動を構成することで問題を解決できます。CUDA* コンピュート能力による制限は、CUDA* ドキュメントの表 (英語) にまとめられています。

ただし、カーネル起動の最大 work-group サイズは、デバイスの潜在的な能力とは必ずしも同じではないため、カーネルが使用するレジスター数を理解する必要があります。大規模な work-group でレジスターの負荷が高くなると、ハードウェアで利用できるレジスター数を超えて、無効なカーネル起動が発生する可能性があります。詳細については、technical-specifications-per-compute-capability (英語) を参照してください。

### work-group の制限超過

work-group の 1 つの次元またはすべての次元の合計サイズ (すべての次元の積) がデバイスでサポートされている最大値を超えるワークを送信すると、`PI_ERROR_INVALID_WORK_GROUP_SIZE` エラーが発生します。同様に、非均一 work-group (`nd_range<1>{48,32}` など) を、サポートしていないデバイス (すべての CUDA* デバイス) に送信すると、同じエラーコードが発生します。これらすべての場合、次のメッセージが出力されます。

```
Non-uniform work-groups are not supported by the target device -54
(PI_ERROR_INVALID_WORK_GROUP_SIZE)
```

現在、work-group が均一であっても、上記の他の制限のいずれかを超える場合は、このエラーが発生します。このメッセージは、プラグインの将来のバージョンで改善される予定です。

1 つの次元またはすべての次元の積が work-group 数の上限を超えると、`CUDA_ERROR_INVALID_VALUE` エラー `PI_ERROR_INVALID_VALUE` にマップされます。例えば、すべての CUDA* デバイスでは、y 次元と z 次元で最大 65535 個の work-group が許可されます。y 次元に 65536 個の work-group を送信すると、以下の例外が発生します。

```
Number of work-groups exceed limit for dimension 1 : 65536 > 65535 -30
(PI_ERROR_INVALID_VALUE)
```

**共有メモリーの最大量超過**

現在、共有メモリーサイズの制限を超えるワークを送信すると、次のエラーが発生します。

```
UR CUDA ERROR:
    Value:           1
    Name:            CUDA_ERROR_INVALID_VALUE
    Description:     invalid argument
    Function:        urEnqueueKernelLaunch
    Source Location: /tmp/tmp.7vgJ2wJCWQ/intel-llvm-
mirror/sycl/plugins/unified_runtime/ur/adapters/cuda/enqueue.cpp:418

Native API failed. Native API returns: -30 (PI_ERROR_INVALID_VALUE) -30
(PI_ERROR_INVALID_VALUE)
```

プラグインの将来のバージョンではエラー処理が改善され、エラーの内容を明確に示す次のメッセージが報告されるようになります。

```
Excessive allocation of local memory on the device
```

**利用可能なレジスター不足**

`CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES` エラーは、CUDA* ブロックごとに使用するレジスターが多すぎることが原因で発生する可能性があります。これは、DPC++ の `PI_ERROR_INVALID_WORK_GROUP_SIZE` エラーコードにマップされ、制限の内容と使用されているレジスター数を示す詳細な例外メッセージが表示されます。

```
Exceeded the number of registers available on the hardware.
    The number registers per work-group cannot exceed 65536 for this kernel on
this device.
    The kernel uses 100 registers per work-item for a total of 1024 work-items
per work-group.
-54 (PI_ERROR_INVALID_WORK_GROUP_SIZE)
```

エラーメッセージ以外に、コンパイル時に `-Xcuda-ptxas --verbose` オプションを指定することで、`ptxas` によってカーネルに割り当てられるレジスター数を簡単にチェックできます。これにより冗長モードが有効になり、バイナリー内のカーネルによるレジスターの使用状況を含むコード生成統計が出力されます。

`ptxas` の詳細出力の例:

```
ptxas info    : Compiling entry function 'my_kernel' for 'sm_75'
ptxas info    : Function properties for my_kernel
    8192 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 100 registers, 256 bytes cmem[0], 1512 bytes cmem[2]
```

最近、DPC++ ランタイムもこのケースを検出して、カーネルがハードウェアで利用可能なレジスター数を超えたことを示す詳細な例外をスローできるようになり、カーネルが実際に使用するレジスター数と失敗した起動構成の work-group に関する情報を提供します。

このタイプのエラーは、DPC++ の ERROR_INVALID_WORK_GROUP_SIZE エラーコードにマップされます。

```
Exceeded the number of registers available on the hardware.
    The number registers per work-group cannot exceed 65536 for this kernel on
this device.
    The kernel uses 100 registers per work-item for a total of 1024 work-items
per work-group.
-54 (PI_ERROR_INVALID_WORK_GROUP_SIZE)
```

カーネルがマルチプロセッサーで使用可能なレジスター数を超えた場合、work-group サイズを縮小すると、CUDA* ブロックで実行されるスレッド数が効果的に削減され、レジスターの負荷を軽減できます。コンパイルされたカーネルの有効な最大 work-group サイズは、SYCL* で次のように照会できます。

```
auto b = sycl::get_kernel_bundle<MyKernel,
sycl::bundle_state::executable>(q.get_context());
auto k = b.template get_kernel<MyKernel>();
auto maxWGSize{k.template
get_info<sycl::info::kernel_device_specific::work_group_size>(q.get_device())}
;
std::cout << "MyKernel max WG size on this device: " << maxWGSize <<
std::endl;
```

ただし、これが望ましい解決策ではない場合、レジスター・プレッシャーを下げて特定のしきい値でスピルが行われるようにコンパイラーに指示することもできます。これにより、スレッドブロックのサイズを小さくすることなく起動を成功させることができます。DPC++ では次のように実現します。

1. ターゲットデバイスの CUDA* アーキテクチャーまたは SM /Compute Capability を指定します。例えば、NVIDIA* GeForce* RTX 3060/TI の場合、次のように指定します。

   ```
   -Xsycl-target-backend --cuda-gpu-arch=sm_86
   ```

2. コンパイルコマンドで -Xcuda-ptxas --maxrregcount=<N> オプションを指定して、カーネル内のレジスターを制限するよう PTX バックエンドに指示します。

**注意:** -Xcuda-ptxas --maxrregcount コンパイラー・オプションによってカーネルのレジスター使用を制限すると、残りのレジスターが DRAM に排出されパフォーマンスに影響する可能性があります。

**プラットフォーム/アーキテクチャー間で移植されたコードの sub-group サイズの問題**

カーネル属性 reqd_sub_group_size を使用して特定の sub-group サイズを設定し、その後、異なるプラットフォームに移植するか、元のアーキテクチャーとは異なるアーキテクチャーで実行されるコードについて考えてみます。このような場合、要求される sub-group サイズがプラットフォーム/アーキテクチャーでサポートされないと、実行時に次のようなエラーがスローされます。

```
Sub-group size x is not supported on the device
```

CUDA* プラットフォームでは、単一の sub-group サイズのみがサポートされるため、次の警告が出力されます。

```
CUDA requires sub_group size 32
```

そして、ランタイムは要求された sub-group サイズに代わって sub-group サイズ 32 を適用します。`reqd_sub_group_size` カーネル属性は、複数の sub-group サイズをサポートするプラットフォーム/アーキテクチャー向けに設計されています。一部の SYCL* コードは、異なる sub-group サイズ間では移植できないことに注意してください。例えば、sub-group 集合の `reduce_over_group` の結果は、sub-group サイズに依存します。異なる sub-group サイズを使用するプラットフォーム/アーキテクチャー間で移植できるコードを作成する場合、次のいずれかを考慮する必要があります。

- 結果が sub-group サイズに依存しないよう、移植可能な方法でコードを記述します。
- コードの sub-group サイズに依存する部分については、sub-group サイズの違いを考慮して、プラットフォーム/アーキテクチャーごとに異なるバージョンを用意します。

### Spir64_gen と NVIDIA*/AMD* ターゲットにして子コンパイルする際の「Could not determine device target」エラー

`nvptx64-nvidia-cuda` または `amdgcn-amd-amdhsa` の後に `-fsycl-targets` フラグで `spir64_gen` ターゲットを指定し、その後 `-Xsycl-target-backend` フラグを使用してターゲット・アーキテクチャーを指定すると、次のようなエラーが発生する可能性があります。

```
Could not determine device target: sm_70.
Error: Cannot get HW Info for device sm_70.
Command was: /usr/bin/ocloc -output /tmp/test-sm_70-7b3cf0-750b2c.out -file
/tmp/icpx-72a9979220/test-sm_70-40eb3b-29e6d1.spv -output_no_suffix -
spirv_input -device sm_70
llvm-foreach:
icpx: error: gen compiler command failed with exit code 223 (use -v to see
invocation)
```

これは、最初に `spir64_gen` ターゲットを指定することによって回避できます。

```
-fsycl-targets=spir64_gen,amdgcn-amd-amdhsa,nvptx64-nvidia-cuda
```

---